ASCII Learning System

アスキー・ラーニングシステム　③応用コース

応用
MS-DOS

村瀬康治 著

改訂新版

アスキー出版局

アスキー・ラーニングシステム

応用MS-DOS

改訂新版

村瀬康治 著

③応用コース

ASCII
LEARNING SYSTEM

アスキー・ラーニングシステム　③応用コース

応用
MS-DOS

村瀬康治 著

改訂新版

アスキー出版局

MS-DOS Learning System 全３巻の構成

アスキー・ラーニングシステムMS-DOS編は、『入門MS-DOS』『実用MS-DOS』『応用MS-DOS』の3部から構成されています。

入門編および実用編では、ビジネスソフトのユーザーをはじめとする、広く一般的なユーザーを対象に、MS-DOSの基礎知識とその活用法を初歩からわかりやすく解説します。また応用編では、MS-DOS上で各種のソフトウェアを開発するプログラマー、あるいはMS-DOSの内部を詳しく知りたいユーザーを対象に、MS-DOSシステムの初歩から、アセンブラやC言語による実際のプログラミング例を豊富にまじえながら、高度な知識に至るまでを解説します。

それぞれの巻は次のように構成されています。

入門MS-DOS：MS-DOS上の各種のビジネスソフトなどを利用しようとする人を対象に、MS-DOSの基本的な機能や操作法をやさしく、ていねいに解説します。とくに、MS-DOSによって初めてOSに接する人も、楽に読み進めるよう十分配慮されています。この入門編により、各種のソフトを活用するためのMS-DOSの知識が、バランスよく得られるようにまとめてあります。

実用MS-DOS：MS-DOSの基礎知識を学んだ人に対して、MS-DOSマシン本来の実力をフルに発揮させ、各種のソフトを効果的に使いこなすための知識とノウハウを提供します。ATOK、松茸、VJEなどによる日本語入力システムの使い方や漢字コードの知識、ハードディスクの各形態のフォーマット処理や初期設定、ハードディスクの効果的運用法、RAMディスクやディスクキャッシュ、それにEMSを実現する具体例とその利用法、さらに各種エディタのやさしい使い方などを実例解説します。また、目的別によるMS-DOSの主要コマンドの使い方を、コマンドリファレンスの形式と実行例により示します。この実用編は、ビジネスソフトのユーザーにも、ソフトウェア開発者にも、MS-DOSマシンを使いこなす重要な知識を提供するでしょう。

応用MS-DOS：MS-DOS上でソフトウェアを開発する人や、MS-DOSの内部構造を詳しく知りたい人に対して、豊富な図や実際のプログラムを示しながら、MS-DOSシステムの構造をその基本から細部まで、ていねいに解説します。また、システムコールの解説や、アセンブラによるソフトウェア開発、さらにC言語の開発環境について解説し、そのプログラミングを実習します。この応用編は、これからMS-DOSシステムを学ぶプログラマーにとって、たいへん親切に書かれた最適の参考書となるでしょう。

このシリーズは、MS-DOSのバージョン3.3をもとに書かれていますが、それ以前のバージョンをお使いの方も、まったく支障なく利用できます。このMS-DOS 3部作を読むにあたり、MS-DOSマシンを操作できれば理想的ですが、いずれも豊富な実行例をもとに解説していますので、紙上体験だけでも大きな効果が得られるでしょう。

はじめに

本書は、MS-DOSの内部を詳しく知りたい、MS-DOSを使ってプログラミングを行ないたいと考えている読者に対して、行く手に立ちはだかる「未知のMS-DOSシステム」や、「実際のプログラミング作法」にアプローチするための最善の「道」を示すとともに、その勇気と希望を与えるものです。そして、MS-DOS付属のプログラマーズマニュアルや、他のさらに高度な専門書に立ち向かうに足る十分な基礎知識と、それらを理解するための多くのヒントを提供します。

本書は、これからの読者に対しては、なぜOSが生まれたのか、なぜシステムコールが存在するのか、また、なぜモジュール別のプログラム開発が必要なのか、…などの基本を、たいへんわかりやすい〝ストーリー〟をもとに、説得力ある解説を行なっています。また、日ごろ疑問に思っている「.COM」と「.EXE」ファイルの違い、ルートディレクトリには収容ファイル数の制限があるのにサブディレクトリにはないこと、パイプやリダイレクトが実現できる仕掛け、…なども、次々と明らかになるでしょう。

さらに経験者に対しては、CP／Mの影響が残された従来のプログラミング形式を払拭し、UNIXやOS/2流の「これからの時代のMS-DOSプログラミング作法」が身に付くよう —— たとえばファイルハンドルの利用や標準入出力／標準エラー出力の利用などを積極的に解説し、例題プログラムに取り入れて実習しています。さらには、MS-DOSの内部構造の解説とともに、アセンブラやC言語などのソフトウェア開発ツールの実際の利用法など、開発現場でまず必要になるノウハウも豊富な実例で解説しています。

さて、本書を含むMS-DOSシリーズ3部作は、初版より6年目を迎え、その発行部数(3部合計)は50万部を超えました。たいへんうれしく思う反面、一層大きな「責任」を感じています。そこでこれを機会に、MS-DOSの進歩や、環境の変化に合せて、3部作を全面的に大改訂いたしました。MS-DOSは今後、UNIXやOS/2の時代を迎えても、かなり長い期間にわたり並行して使われていく見とおしです。本書の改訂の主旨は、ただMS-DOSのバージョンアップに対応しただけでなく、さきに述べたように、「これからの時代におけるMS-DOSのプログラミング」を鮮明に打ち出した点にあります。

なお、本書の大改訂にあたっては、初版時に全面的に協力していただいた、『はじめて読むMASM』などでおなじみの、蒲地輝尚さんにたいへんお世話になりました。彼の援助がなかったら、これほど親切で詳細・緻密なMS-DOSの解説書は、決して完成しなかったでしょう。蒲地さんと、アスキーのスッタフ集団の総力のおかげで、ほんとうにすばらしい解説書に仕上がったと感謝しています。どうもありがとうございました。

1989年10月　　村瀬康治

目　次

MS-DOS Learning System 全３巻の構成 ―― 2
はじめに ―― 3

1章　ユーザープログラム作成実習

1.1　リードオンリーおよび隠しファイルについて ―― 11
1.2　作成するプログラムの機能 ―― 12
1.3　実行可能なプログラム作成作業 ―― 13
1.4　CHMOD プログラムの実行と応用 ―― 25
1.5　本章は MS-DOS 解説のためのプロローグ ―― 28

2章　MS-DOS のファイルシステム

2.1　ファイルの読み出し／書き込み ―― 31
　2.1.1　ファイルの読み出しプログラムの作成 ―― 31
　　■ファイルのオープン ……35
　　■ファイルの読み出し(リード) ……39
　　■ディスク・バッファリング ……42
　　■ファイルのクローズ ……44
　2.1.2　ファイルの書き込みプログラムの作成 ―― 44
　　■ファイルのクリエイト(新規ファイルの作成) ……48
　　■ファイルへの書き込み(ライト) ……51
　　■書き込み後のファイルのクローズ ……55
2.2　階層ディレクトリの仕組み ―― 58
　2.2.1　サブディレクトリの仕組み ―― 58
　2.2.2　「.」と「..」 ―― 63
　2.2.3　ディレクトリのバッファリング ―― 63
　　■アクセス途中でのディスクの交換 ……64
　　■バッファの数 ……66
2.3　ファイルアクセスのメカニズム ―― 67
　2.3.1　ディスクの物理フォーマット ―― 67
　2.3.2　ディスク上の領域 ―― 68
　2.3.3　MS-DOS 標準ディスクフォーマット ―― 69
　2.3.4　デバッガによる直接ディスクアクセス ―― 71
　2.3.5　FAT を見る ―― 72
　2.3.6　ディレクトリを見る ―― 73
　2.3.7　ファイルの属性(アトリビュート)を見る ―― 75
　2.3.8　ファイル属性を直接書き換える ―― 77
　2.3.9　目的のファイルの中身はどのセクタに? ―― 80
　　■16 ビット FAT　ーハードディスクを有効に活用するための機能ー ……89
　2.3.10　消去したファイルの行方は? ―― 91
　2.3.11　サブディレクトリ内の消去されたファイルの復活 ―― 99

3章　MS-DOSの仕組みと働き

3.1　OS(オペレーティングシステム)とは ―― 107

- 3.1.1　OSのないコンピュータシステムを使うと ―― 107
- 3.1.2　OSの概念の誕生 ―― 110
- 3.1.3　ライブラリの誕生 ―― 112
- 3.1.4　ハードウェアに依存する部分の独立 ―― 113
- 3.1.5　ユーザープログラムの各機種間での互換性 ―― 114
- 3.1.6　ソフトウェア開発環境 ―― 116
- 3.1.7　パーソナル・コンピュータ用OSの誕生とMS-DOS ―― 116
- 3.1.8　MS-DOSの構造の予備知識 ―― 117

3.2　MS-DOSの起動の仕組み ―― 118

- 3.2.1　IPLによる2つのシステムファイルのロード ―― 118
- 3.2.2　ハードウェアとMS-DOSの初期化 ―― 120
- 3.2.3　CONFIG.SYSファイルによるシステムのインストール ―― 121
  - ■CONFIG.SYSファイルに登録できるおもなコマンド ……122
- 3.2.4　COMMAND.COMのロード ―― 124
- 3.2.5　COMMAND.COMの起動 ―― 125

3.3　MS-DOSの基本構成 ―― 125

3.4　内蔵コマンド実行時の各部の働き ―― 127

- 3.4.1　COMMAND.COM ―― 127
  - ■COMMAND.COMはBASICのプログラム? ……128
- 3.4.2　MSDOS.SYS ―― 131
- 3.4.3　IO.SYS ―― 132

3.5　外部プログラム実行時の各部の働き ―― 133

3.6　COMMAND.COMの働き ―― 134

- 3.6.1　外部プログラムの実行処理 ―― 134
- 3.6.2　標準入出力 ―― 136
  - ■リダイレクト ……136　■パイプ ……139　■フィルタ ……140
- 3.6.3　内蔵コマンドの実行処理 ―― 144
  - ■環境変数の設定 ……145
- 3.6.4　バッチコマンドの処理 ―― 146

3.7　MSDOS.SYSの働き ―― 147

- 3.7.1　外部プログラムの実行とMSDOS.SYS ―― 148
  - ■プロセス管理 ……148　■メモリ管理 ……150
  - ■PSP(プログラムセグメント・プレフィックス) ……151
  - ■環境文字列の形式 ……153　■オブジェクト形式 ……154
- 3.7.2　システムコールとソフトウェア割り込み ―― 156

3.8 IO.SYS の働き ― 158
- 3.8.1 デバイスドライバ ― 158
- 3.8.2 ブロック型デバイスとキャラクタ型デバイス ― 160
- 3.8.3 標準デバイスドライバ ― 162
- 3.8.4 デバイスファイル（キャラクタ型デバイス） ― 163
- 3.8.5 ドライブ（ブロック型デバイス） ― 164
- 3.8.6 デバイスドライバとの入出力 ― 165
- 3.8.7 BPB とメディアチェック ― 166
- 3.8.8 CONFIG.SYS ファイルによるデバイスドライバの登録 ― 168
- 3.8.9 ADDDRV、DELDRV コマンドによるデバイスドライバの登録と削除 ― 170
- 3.8.10 デバイスドライバを作成するメリット ― 173

4章 システムコールとソフトウェア割り込み

4.1 システムコールとファンクションリクエストの概念 ― 177

4.2 ファンクションリクエストの使い方の基礎知識 ― 180
- ■目的のファンクションをコールする手順 …… 182
- ■ファンクションリクエストの結果について …… 183
- ■ファンクションリクエスト実行時のエラー …… 185
- ■ファンクションリクエストによるほかのレジスタおよびスタックポインタへの影響 …… 187

4.3 システムコール ― 189
- ■システムコールを使った応用プログラム例 …… 191

4.4 主要ファンクションリクエスト実例集 ― 196
- ■ファンクション 05_H、$0A_H$、$4C_H$を利用したプログラム例 …… 197
  （バッファードキーボード入力とプリンタへの出力プログラム BUFFPRN）
- ■ファンクション $3D_H$、$3F_H$、$3E_H$、02_Hを利用したプログラム例 …… 203
  （2 章のファイルの読み出しプログラム FREAD）
- ■ファンクション $3C_H$、40_H、$3E_H$、42_Hを利用したプログラム例 …… 205
  （2 章のファイルへの書き込みプログラム FWRITE）
- ■ファンクション 40_Hを利用したプログラム例 …… 211
  （標準出力を利用した FREAD プログラム）
- ■ファンクション $3D_H$、$3E_H$、$3F_H$、40_Hを利用したプログラム …… 215
  （キーボード入力とプリンタへの出力プログラム LPRN）
- ■ファンクション 43_Hを利用したプログラム例 …… 220
  （1 章のファイル属性設定プログラム CHMOD）
- ■ファンクション $4A_H$、$4B_H$を利用したプログラム例 …… 222
  （子プロセスの起動プログラム）

5章　アセンブラによるソフトウェア開発

5.1　マクロアセンブラとは ―― 229
- ■リロケート機能 ……230　■マクロ機能 ……234
- ■モジュール別プログラム開発法 ……240

5.2　アセンブラによるソフトウェア開発の手順 ―― 245
- ■COM 形式と EXE 形式 ……248　■モジュール別プログラミングの実習 ……248

5.3　アセンブリ・ソースファイルの書き方 ―― 251

5.3.1　擬似命令の使い方 ―― 251
- ■SEGMENT……251　■STACK ……252　■ASSUME ……252
- ■ORG ……253　■END ……253　■PROC ……254
- ■PUBLIC、EXTRN ……254

5.3.2　オブジェクト形式別ソースファイルの書き方 ―― 256

5.3.3　16 進電卓プログラムのソースファイルの作成 ―― 260

5.4　アセンブル ―― 267
- ■オプションスイッチ ……269

5.5　リンク ―― 269
- ■オブジェクト形式の変換 ……270

5.6　デバッグ ―― 271
- ■シンボリックデバッガ ……280　■ソースコードデバッガ ……283

5.7　ライブラリの活用 ―― 283

6章　C 言語によるプログラム開発

6.1　C 言語の世界 ―― 289
- ■C 言語はアセンブラだ ……289　■C プログラムの入り口と出口 ……290
- ■書式変換 ……292　■プログラム実行中のレジスタ内容の表示 ……293
- ■システムコールの直接呼び出し ……293　■ライブラリあっての C 言語 ……294
- ■C 言語の各処理系間の互換性 ……295　■C 言語のエラーチェック ……295

6.2　C 言語と MS-DOS との相性 ―― 296
- ■ファイルのオープン／クローズ、読み出し／書き込み ……296
- ■バッファリング ……297　■復帰改行コードの相違についての注意 ……298
- ■その他の関数 ……299　■ASCIZ 文字列の役割 ……299

6.3　大きなプログラムの開発 ―― 299
- ■モジュール別プログラミング ……300　■MAKE……301

6.4　C 言語によるプログラム作成実習 ―― 305
- ■ファイル属性変更プログラムの作成 ……305

6.5　より進んだプログラミング環境 ―― 313
- ■統合化されたプログラミング環境 ……313　■ソースコードデバッガ　CODEVIEW ……314

7章　デバイスドライバの作成とマウスの応用

7．1　作成する「お絵かきプログラム」の構成――319
7．2　デバイスドライバを利用したお絵かきプログラムの作成――321
　7．2．1　ドットを打つデバイスドライバの作成――321
　　■作成するデバイスドライバの仕様……322
　　■デバイスドライバ PLOT のソースファイルの作成……324
　　■デバイスドライバ PLOT の動作確認……331
　7．2．2　デバイスドライバの内部解説――333
　　■デバイスドライバの実行可能ファイル形式……333
　　■デバイスヘッダ……333
　7．2．3　デバイスドライバのアクセス法――339
　7．2．4　デバイスドライバ PLOT を利用するユーザープログラムの作成――339
7．3　アセンブラによるお絵かきプログラムの作成――346
7．4　デバイスドライバを使わないお絵かきプログラムの作成――349
7．5　デバイスドライバとユーザープログラムの相違――358

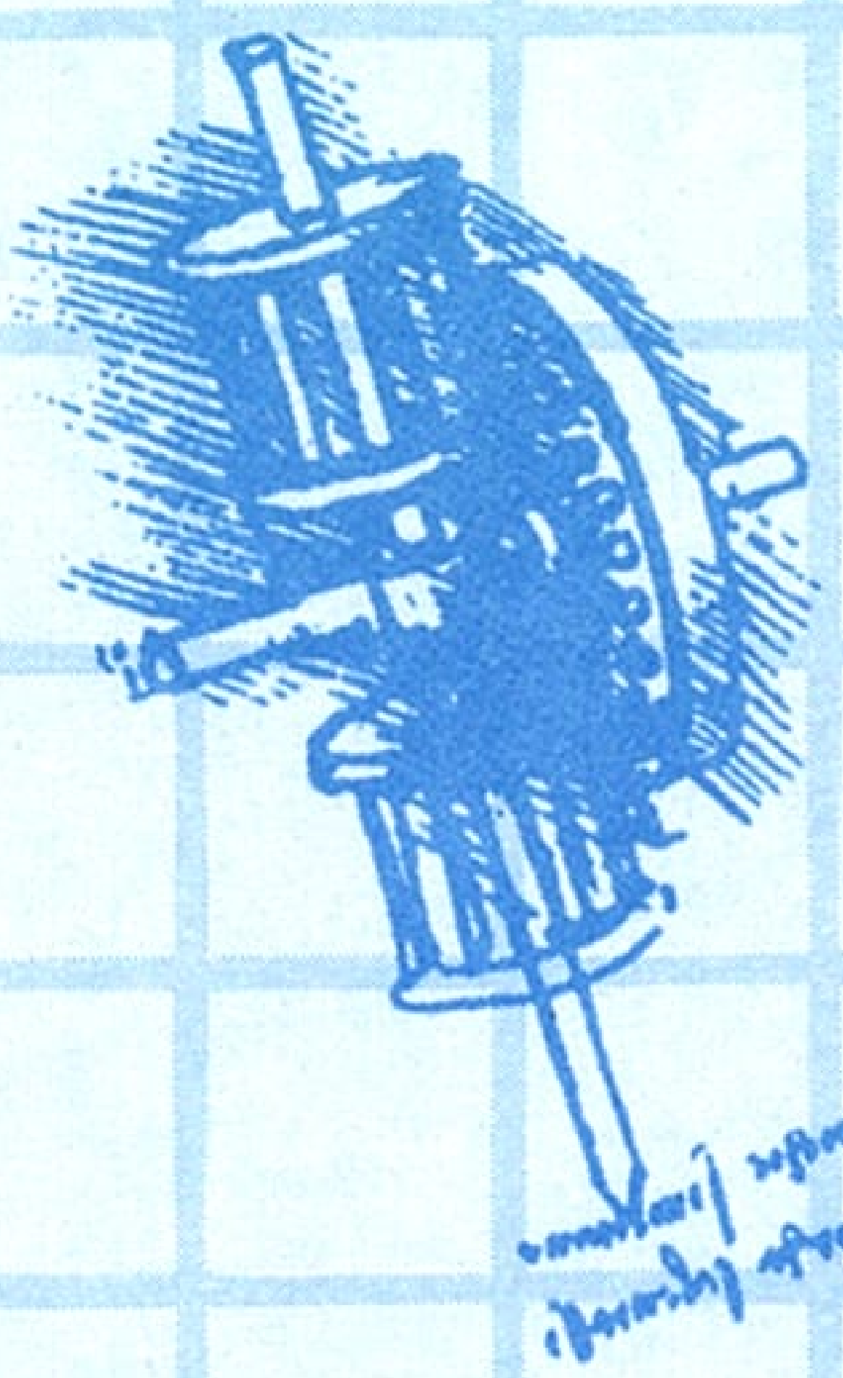
APPENDIX

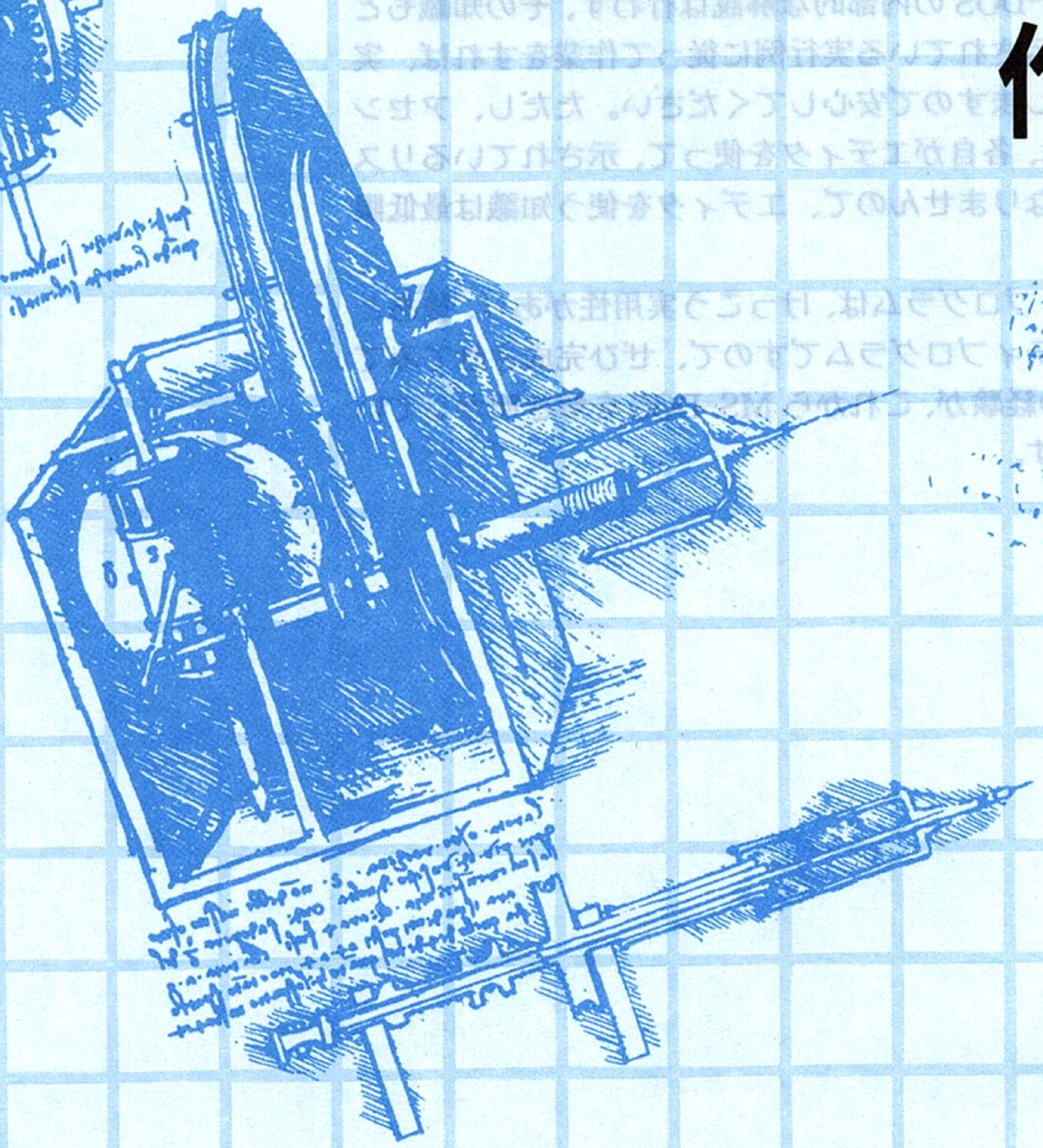
数字羅列ファイルを作成するプログラム「mk012」――360
大文字↔小文字変換プログラム「ulconv」――360

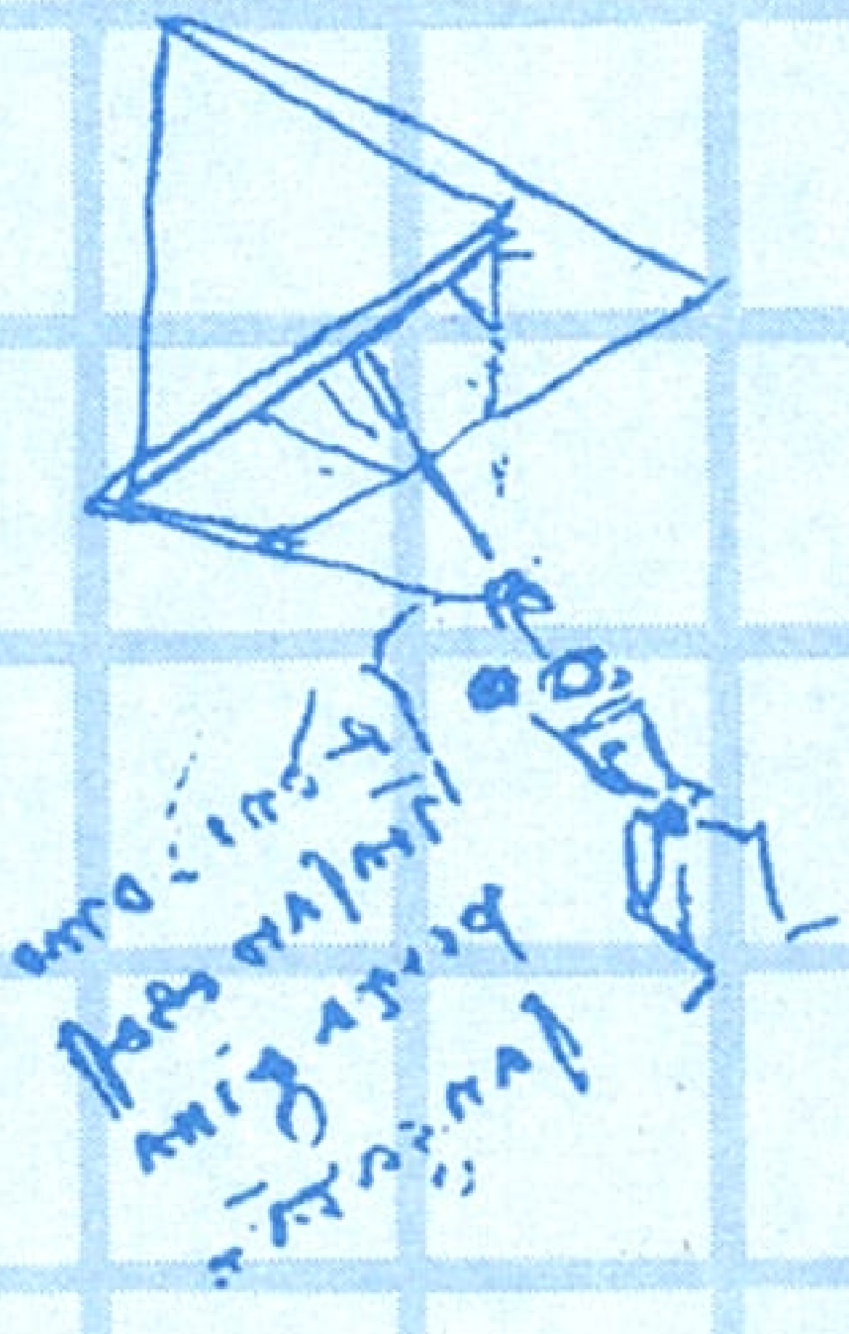
おわりに――362
索　引――363

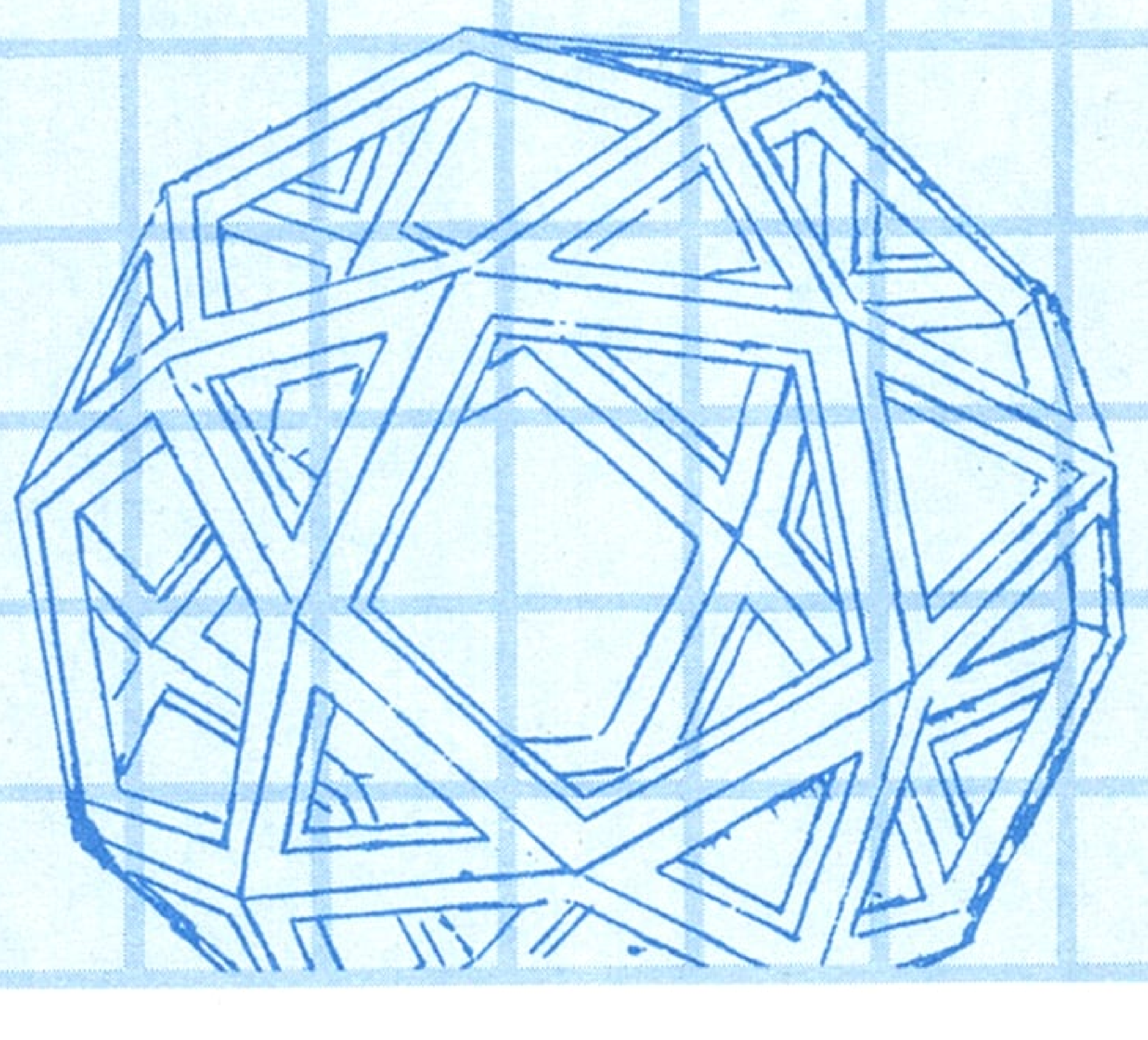
本文イラスト：長谷川 弘
扉イラスト：シグマCC㈱

1章
ユーザープログラム作成実習



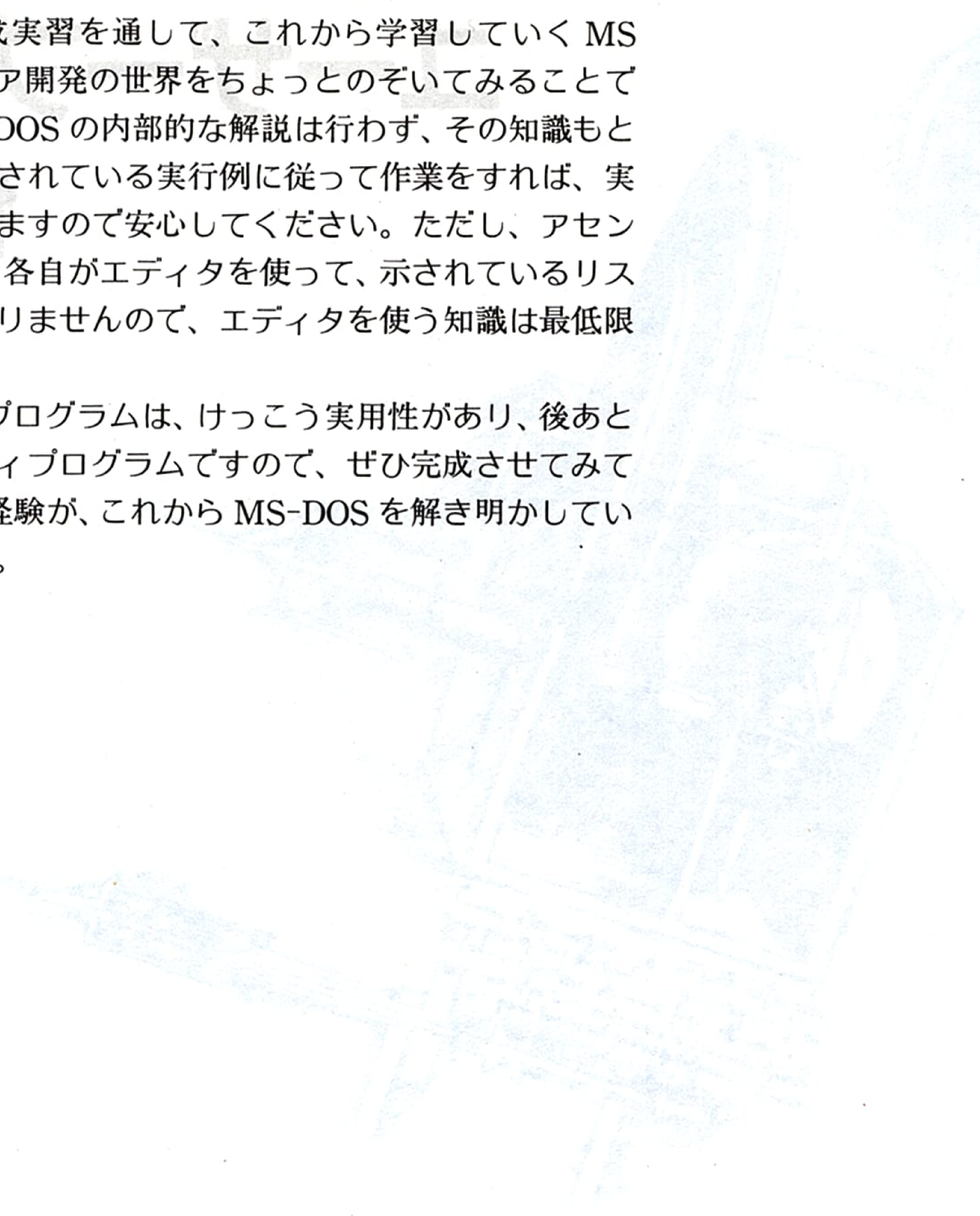






本章では、まずCHMOD(チェンジモード)と名付けた、簡単な実用プログラムをアセンブリ言語(アセンブラ)で作成してみましょう。ここでの狙いは、このプログラムの作成実習を通して、これから学習していくMS-DOSの内部や、ソフトウェア開発の世界をちょっとのぞいてみることです。ですから、本章ではMS-DOSの内部的な解説は行わず、その知識もとりあえず必要ありません。示されている実行例に従って作業をすれば、実行可能なプログラムが完成しますので安心してください。ただし、アセンブリ・ソースファイルだけは、各自がエディタを使って、示されているリストどおりに作成しなければなりませんので、エディタを使う知識は最低限必要になります。

ここで作成するCHMODプログラムは、けっこう実用性があり、後あと何かと役に立つユーティリティプログラムですので、ぜひ完成させてみてください。何よりもここでの経験が、これからMS-DOSを解き明かしていくための糸口となるはずです。



1.1 リードオンリーおよび隠しファイルについて

MS-DOSのシステムディスクには、任意のファイルを消去／更新できない**リードオンリーファイル**(read only ファイル)にするATTRIBコマンド*が用意されていますが、DIRコマンドでは表示できない**隠しファイル**(hidden ファイル)にすることはできません。ところがMS-DOS内部のファイルシステムには、これらを含め、ファイルのあらゆる属性(ファイルアトリビュート)を設定する機能があり、これをユーザープログラムで利用する手段が提供されています。したがって、この機能を利用したユーザープログラムを作成すれば、ファイルの属性を自由に変更することが可能になります。

ファイル属性とは、個々のファイルが持つファイルの性質や種類を分類するグループ名のことです。MS-DOSのファイル属性のおもなものを**表1.1**に示します。

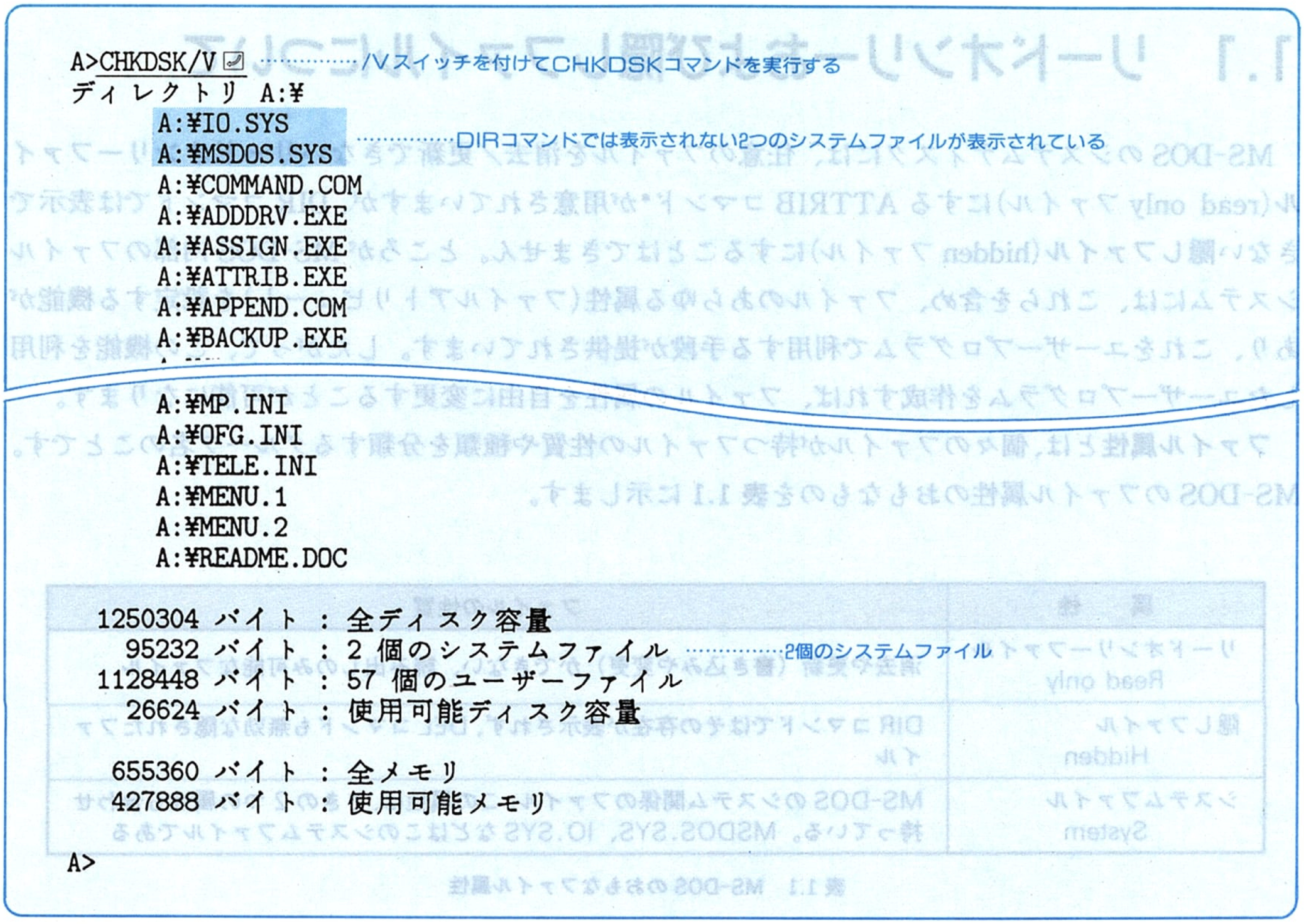
属　性	ファイルの性質
リードオンリーファイル Read only	消去や更新（書き込みや変更）ができない、読み出しのみ可能なファイル
隠しファイル Hidden	DIRコマンドではその存在が表示されず、DELコマンドも無効な隠されたファイル
システムファイル System	MS-DOSのシステム関係のファイル。この属性は、さきの2つの属性も合わせ持っている。MSDOS.SYS、IO.SYSなどはこのシステムファイルである

表1.1　MS-DOSのおもなファイル属性

MS-DOSのファイル属性の種類はこのほかにもありますが、代表的なものは以上です。また、**通常のファイル**とは、これらの属性が付かない、つまりDIRコマンドで表示されたりDELコマンドで削除されたりする、リード／ライト可能なファイルのことです(属性については2章で詳しく解説する)。

この3つの属性の中で、システムファイル(隠しファイルであり、かつリードオンリーファイルでもある)については、MS-DOSのシステムディスクが手元にあれば、**図1.1**ように簡単にその実体を見ることができます。

＊ ATTRIBは、MS-DOSバージョン3.*x*から追加された外部コマンドである。



```
A>CHKDSK/V↵
ディレクトリ A:¥
      A:¥IO.SYS
      A:¥MSDOS.SYS
      A:¥COMMAND.COM
      A:¥ADDDRV.EXE
      A:¥ASSIGN.EXE
      A:¥ATTRIB.EXE
      A:¥APPEND.COM
      A:¥BACKUP.EXE

      A:¥MP.INI
      A:¥OFG.INI
      A:¥TELE.INI
      A:¥MENU.1
      A:¥MENU.2
      A:¥README.DOC

  1250304 バイト : 全ディスク容量
    95232 バイト : 2 個のシステムファイル
  1128448 バイト : 57 個のユーザーファイル
    26624 バイト : 使用可能ディスク容量

   655360 バイト : 全メモリ
   427888 バイト : 使用可能メモリ
A>
```

/Vスイッチを付けてCHKDSKコマンドを実行する

DIRコマンドでは表示されない2つのシステムファイルが表示されている

2個のシステムファイル

図 1.1　システムファイルを見る

さて本章では、任意のファイルにこれらのファイル属性を設定する(つまり、任意のファイルをそのような性質にする)プログラムを作成しましょう。

1.2　作成するプログラムの機能

ここで作成するプログラムは、任意のファイルをリードオンリーファイルにしたり、隠しファイルにしたり、またそれらを通常のファイルに戻したりする機能を持っています。プログラムを簡略にするために、システムファイルについてはサポートしていませんが(必要ならば、のちほど各自で容易に追加できるでしょう)、十分実用的なものです。

作成するプログラム名を「CHMOD」(チェンジモード)とすることにして、まずこのプログラムを実行する際のコマンドの与え方と、それぞれの機能を示します。

- **CHMOD　ファイル名/?**………「ファイル名」で指定されるファイルの現在の属性を表示する
- **CHMOD　ファイル名/R**………「ファイル名」で指定されるファイルをリードオンリーファイルにする
- **CHMOD　ファイル名/W**……リード／ライト可能な通常のファイルにする
- **CHMOD　ファイル名/H**……隠しファイルにする
- **CHMOD　ファイル名/N**……隠しファイルを見えるファイルにする

CHMODプログラムはこのような機能を持っていますので、日常有効に利用できると思います。では、プログラムの作成作業に移りましょう。

1.3　実行可能なプログラム作成作業

本章で作成するCHMODプログラムは、アセンブラを使って開発します。そのためにはまず、次に示す各種のソフトウェア開発ツール(ソフトウェアを開発するために必要な道具としてのソフトウェア)が必要です。以降の実習のために、事前に用意しておいてください。

- **任意のエディタ**…………………(システムディスクにはEDLIN.EXE(またはCOM)が含まれているが、多くの市販品がある)
- **MASM.EXE***……………………マクロアセンブラ
- **LINK.EXE**…………………………リンカ
- **EXE2BIN.EXE**…………………実行可能形式のEXEファイルから、もう1つの実行可能形式であるCOMファイルに変換するプログラム

これらのツールがそろったら、**図1.2**のような手順で作業を進めます。通常のソフトウェア開発では、必ずデバッグ作業が伴いますが、ここでは正しいアセンブリ・ソースプログラムが提供されていますので、デバッグ作業は省略できます(なお、アセンブラによるソフトウェア開発について、詳しくは5章で解説する)。

＊ MASMは、バージョン2.*x*以前のMS-DOSには、システムディスクに含まれていたが、現在は別売となっているので、市販の「マイクロソフトマクロアセンブラ」(MASM)を別途購入する必要がある。

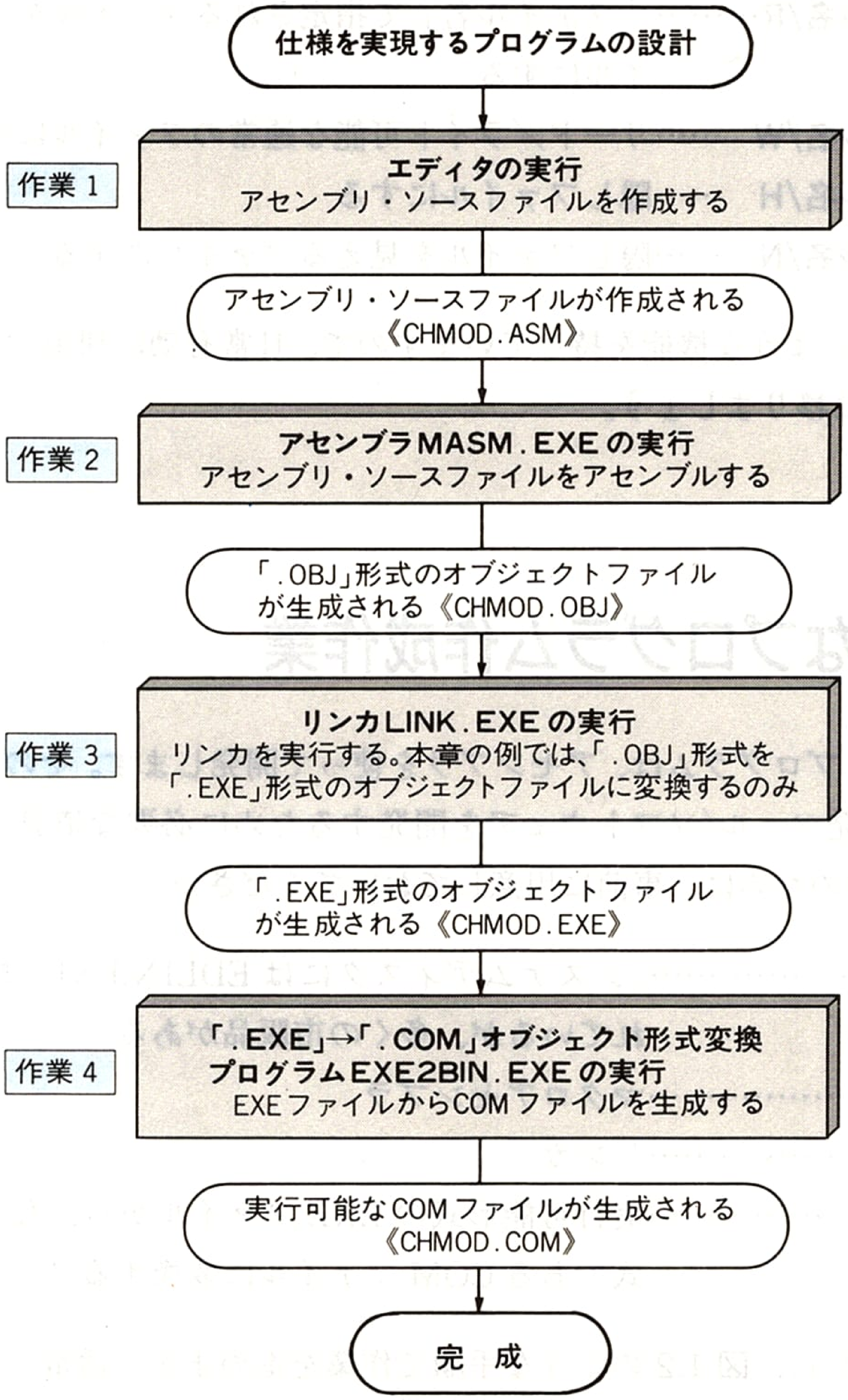

仕様を実現するプログラムの設計
作業 1
エディタの実行
アセンブリ・ソースファイルを作成する
アセンブリ・ソースファイルが作成される
《CHMOD.ASM》
作業 2
アセンブラMASM.EXEの実行
アセンブリ・ソースファイルをアセンブルする
「.OBJ」形式のオブジェクトファイル
が生成される《CHMOD.OBJ》
作業 3
リンカLINK.EXEの実行
リンカを実行する。本章の例では、「.OBJ」形式を
「.EXE」形式のオブジェクトファイルに変換するのみ
「.EXE」形式のオブジェクトファイル
が生成される《CHMOD.EXE》
作業 4
「.EXE」→「.COM」オブジェクト形式変換
プログラムEXE2BIN.EXEの実行
EXEファイルからCOMファイルを生成する
実行可能なCOMファイルが生成される
《CHMOD.COM》
完　成

図 1.2　本章におけるアセンブラによるソフトウェア開発の作業手順

作業１　ソースファイルの作成

まず、CHMODプログラムのアセンブリ・ソースプログラムを示します（**リスト1.1**）。みなさんが行う最初の作業は、エディタを使って、このリストどおりのファイルをディスク上に作成することです。ファイル名は「CHMOD.ASM」としてください。ファイル名の「CHMOD」の部分は、もしこの名前が気に入らなければほかのものに変更してもかまいませんが、「.ASM」の部分は通常「.ASM」としておきます（ほかのものでもアセンブルは可能であるが、コマンドラインの記述がめんどうになる）。

```
;; CHMOD.ASM    CHANGE FILE ATTRIBUTES
;;
;;      09H     Display String
;;      43H     Change Attributes
;;      4CH     Terminate a Process
```

＊このプログラムのさらに詳しい解説は、4章のリスト4.8で行っていますので、ここでは概要を示します。

```
PRINT   MACRO   MSGADR
        MOV     AH,09H
        MOV     DX,OFFSET MSGADR
        INT     21H
        ENDM
```

……………文字列を表示するファンクションリクエスト

```
CSEG    SEGMENT
        ASSUME  CS:CSEG,DS:CSEG,ES:CSEG

        ORG     80H
CMDLEN  DB      ?
CMDBUF  DB      127 DUP (?)

        ORG     100H
START:
```

←……………CHMODプログラムは、ここから実行が開始される

```
        MOV     BL,CMDLEN
        CMP     BL,4
        JB      ERROR
        XOR     BH,BH
        CMP     CMDBUF[BX-2],'/'
        JNE     ERROR
        MOV     DX,OFFSET CMDBUF[1]
        MOV     CMDBUF[BX-2],0
```

入力されたコマンドが正しいかどうかのチェックと前処理

```
        MOV     AX,4300H
        INT     21H
```

……………指定されたファイルの属性を得るファンクションリクエスト。ファイル名を指すDXは前処理で設定済

```
        JC      ERROR
        TEST    CX,00011000B
        JNE     ERROR
```

指定されたファイルが存在しなかったり、ディレクトリやボリュームラベルであった場合は、「ファイルなし」エラーとする

— リスト1.1 —（次ページ以下に続く）

```
GETSW:
          MOV      AL,CMDBUF[BX-1]     ┐
          CMP      AL,’?’              │
          JE       ?SWITCH             │
          AND      AL,NOT(’A’ XOR ’a’) │
          CMP      AL,’R’              │
          JE       RSWITCH             │ 指定されたスイッチ「/x」を識別し、それぞれの処理へジャンプする
          CMP      AL,’W’              │
          JE       WSWITCH             │
          CMP      AL,’H’              │
          JE       HSWITCH             │
          CMP      AL,’N’              │
          JE       NSWITCH             ┘
          JMP      SHORT ERROR ……………誤ったスイッチを指定した場合は「スイッチの不正」エラーとする

RSWITCH:
          OR       CX,00000001B  ┐
          JMP      SHORT CHANGE  ┘ リードオンリーファイルにする前準備
WSWITCH:
          AND      CX,11111110B  ┐
          JMP      SHORT CHANGE  ┘ リード／ライトファイルにする前準備
HSWITCH:
          OR       CX,00000010B  ┐
          JMP      SHORT CHANGE  ┘ 隠しファイルにする前準備
NSWITCH:
          AND      CX,11111101B ……………見えるファイルにする前準備
CHANGE:
          MOV      AX,4301H
          INT      21H     ……………上で準備したそれぞれの属性ファイルに設定するファンクションリクエスト
          JC       ERROR
          XOR      AL,AL ……………正常終了のコードを設定する
          JMP      SHORT RETURN

ERROR:
          PRINT    ERRMSG  ┐ エラーメッセージを出力してエラーコードを設定する
          MOV      AL,1    ┘
RETURN:
          MOV      AH,4CH  ┐ CHMODプログラムを終了する
          INT      21H     ┘

?SWITCH:
          TEST     CX,00000001B   ┐
          JE       RWFILE         │
          PRINT    ROMSG          │
          JMP      SHORT HDCHK    │
RWFILE:                           │
          PRINT    RWMSG          │
HDCHK:                            │ /?スイッチの処理
          TEST     CX,00000010B   │ さきに得られたファイルの属性の値によって、
          JE       SYSCHK         │ それぞれの属性のメッセージを表示する
          PRINT    HDMSG          ┘
```

```
SYSCHK:
          TEST      CX,00000100B
          JE        CHKEND
          PRINT     SYSMSG
CHKEND:
          PRINT     FILEMSG
          XOR       AL,AL
          JMP       SHORT RETURN

ERRMSG    DB        ODH,OAH,"Illegal switch or filename",ODH,OAH,'$'
RWMSG     DB        ODH,OAH,"Read/Write ",'$'
ROMSG     DB        ODH,OAH,"Read Only ",'$'
HDMSG     DB        "Hidden ",'$'
SYSMSG    DB        "System ",'$'
FILEMSG   DB        "File",ODH,OAH,'$'

CSEG      ENDS
          END       START
```

…………正常終了のコードを設定する

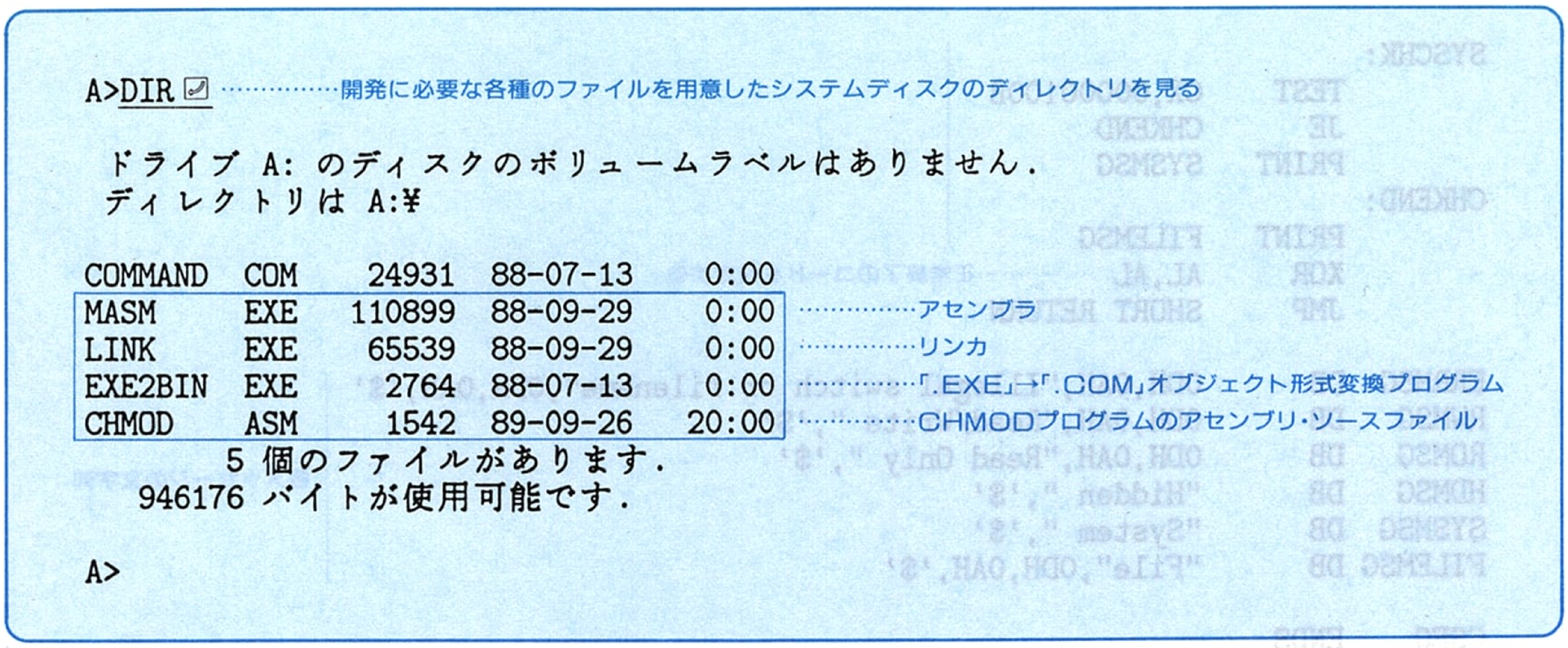
各メッセージの文字列

リスト 1.1　CHMOD プログラムのアセンブリ・ソースプログラム CHMOD.ASM

このプログラムは、MS-DOS 内部に用意されている各種の機能を、ユーザープログラムで利用するための、**システムコール**（ファンクションリクエスト）と呼ばれる基本機能を利用して作られています。キーボードから文字を入力する、ディスプレイにメッセージを表示する、ファイル属性を変更するなど、これらのすべてはシステムコールによって実現されているのです（システムコールについて詳しくは、3.7.2 および 4 章で解説する）。

さて、このプログラムのソースファイルを作成しなければなりませんが、エディタが使えない方は、残念ですが本章の実習ができませんので、この際思い切ってエディタの使い方をマスターしましょう（ここでは初歩的な使い方ができればよい。また、小さなソースファイルであれば、ワープロをエディタの代わりに使う手もある）。MS-DOS のシステムディスクに含まれているエディタ EDLIN や市販のスクリーンエディタである Mifes-98 の基本的な使い方、それにワープロをエディタ代わりに使う方法などは、『実用 MS-DOS』に紹介してあります。

ここで作成されたアセンブリ・ソースファイルと開発ツールを合わせて、以降の実習には、**図 1.3** に示す 4 つのファイルがディスク上に用意されていなければなりません。確認しておいてください。



```
A>DIR↵

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13     0:00
MASM     EXE    110899  88-09-29     0:00
LINK     EXE     65539  88-09-29     0:00
EXE2BIN  EXE      2764  88-07-13     0:00
CHMOD    ASM      1542  89-09-26    20:00
        5 個のファイルがあります.
   946176 バイトが使用可能です.

A>
```

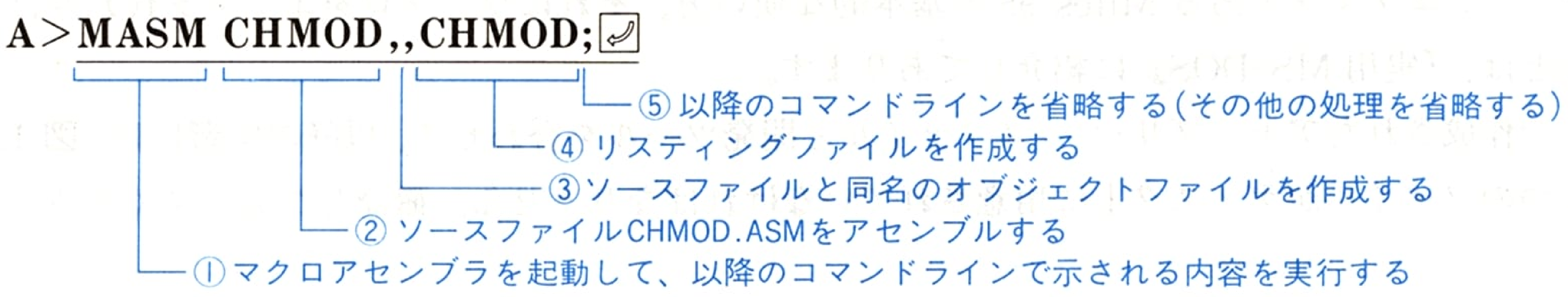
……………開発に必要な各種のファイルを用意したシステムディスクのディレクトリを見る
……………アセンブラ
……………リンカ
……………「.EXE」→「.COM」オブジェクト形式変換プログラム
……………CHMODプログラムのアセンブリ・ソースファイル

図 1.3　実習に必要なソースファイルおよび開発ツール

作業２　アセンブル

エディタを使って、リスト1.1に示したアセンブリ・ソースファイルを、ファイル名CHMOD.ASMとして作成したならば、ミスタイプなどの誤りがないかをよくチェックしたあと、アセンブルします。

ソースファイルをアセンブルするには、マクロアセンブラ **MASM.EXE** を使います。アセンブラを実行する際には、生成するファイルの種類の指定や、そのファイル名の指定などを行うためのいくつかのコマンド形式があり、また、それらを対話形式で行うこともできますが、ここではなるべく簡単なコマンド形式で実行することにしましょう。ソースファイルのファイル名がCHMOD.ASMですので、**図1.4** のコマンドラインを実行します(MASMに関して、詳しくは5章参照)。

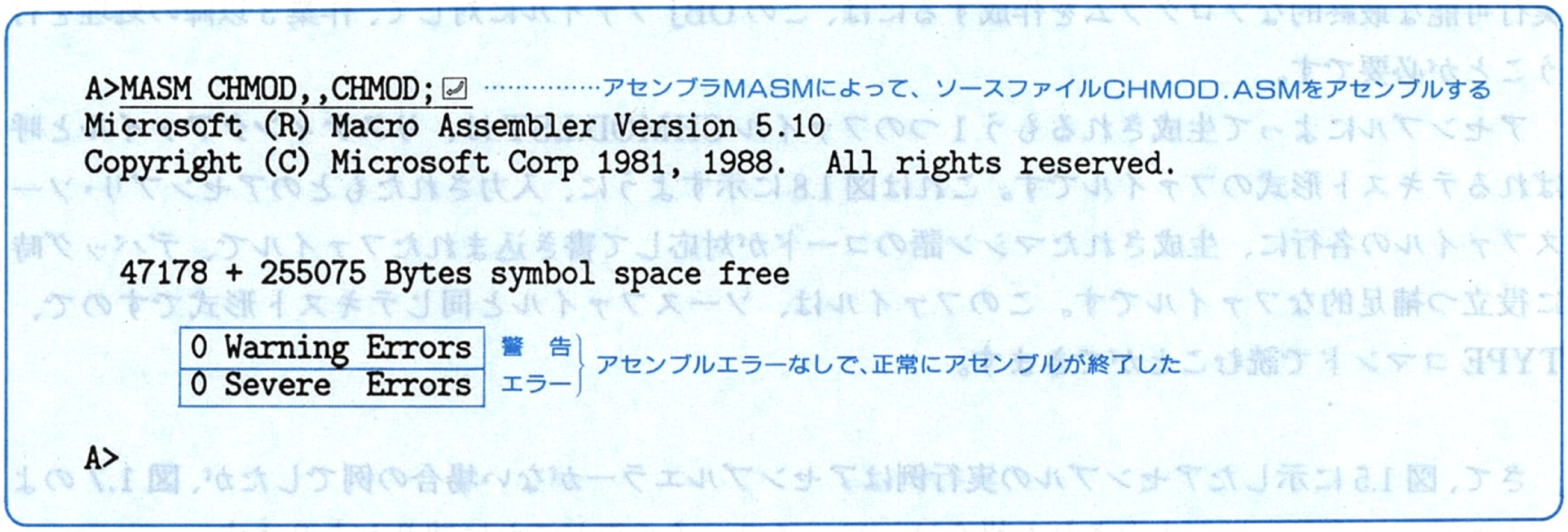


```
A>MASM CHMOD,,CHMOD;↵
```

⑤以降のコマンドラインを省略する(その他の処理を省略する)
④リスティングファイルを作成する
③ソースファイルと同名のオブジェクトファイルを作成する
②ソースファイルCHMOD.ASMをアセンブルする
①マクロアセンブラを起動して、以降のコマンドラインで示される内容を実行する

図 1.4　アセンブラを実行するコマンドライン

このコマンド形式によるアセンブルの実行例を図 **1.5** に示します。



```
A>MASM CHMOD,,CHMOD;↵
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


  47178 + 255075 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

A>
```

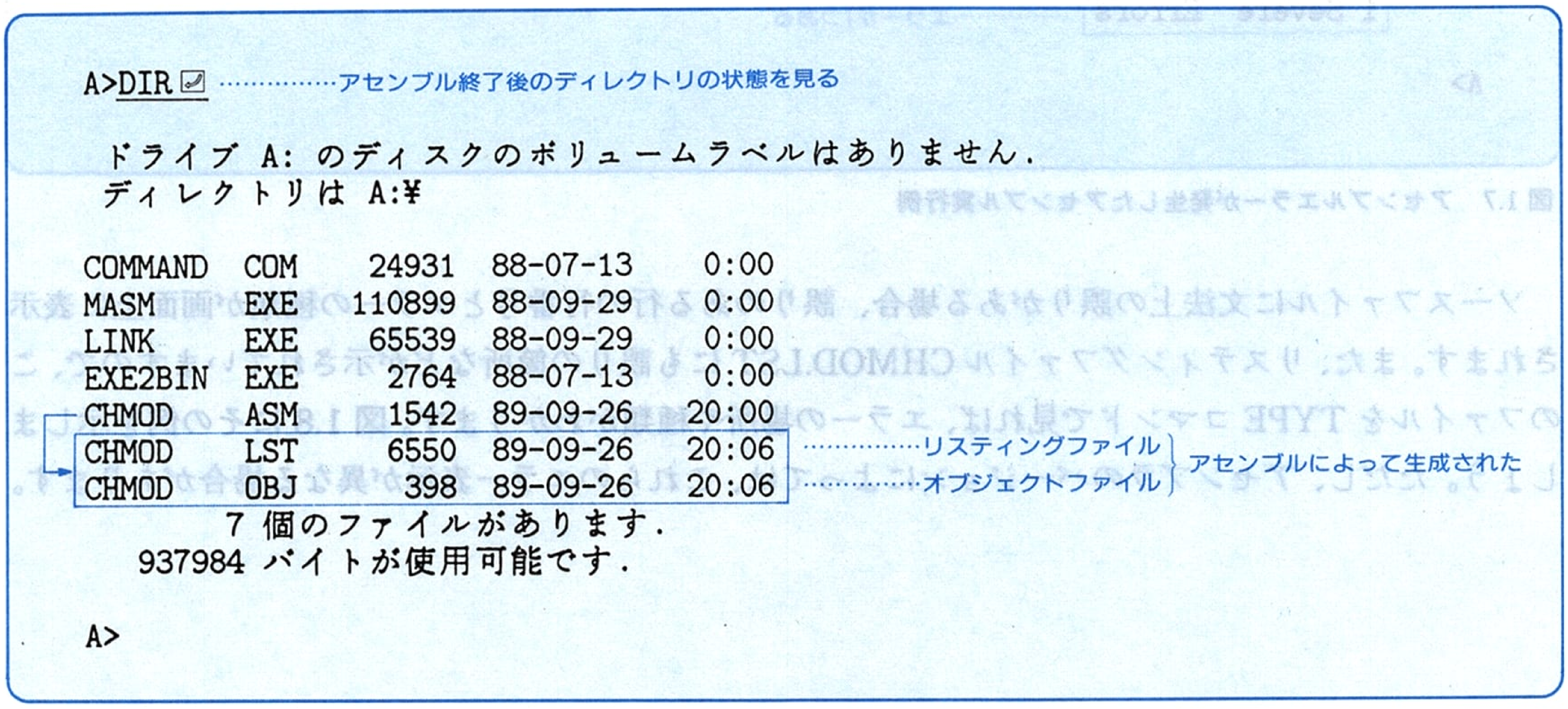
アセンブラMASMによって、ソースファイルCHMOD.ASMをアセンブルする

警　告
エラー
アセンブルエラーなしで、正常にアセンブルが終了した

図 1.5　アセンブルの実行例(アセンブルエラーなしの場合)

アセンブリ・ソースファイルに文法上の誤りがなければ、この例のように「エラーなし」でアセンブルが終了します。ただし、アセンブル時のエラー(アセンブルエラー)が0だからといって、そのプログラムにバグがなく正常に動作するという保証があるわけではありません。あくまでアセンブルする際のエラーがなかったというだけのことです。

アセンブルが終了すると「CHMOD.OBJ」および「CHMOD.LST」の2つのファイルが生成されています。この様子をDIRコマンドで見てみましょう(図 **1.6**)。



```
A>DIR↵

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13    0:00
MASM     EXE    110899  88-09-29    0:00
LINK     EXE     65539  88-09-29    0:00
EXE2BIN  EXE      2764  88-07-13    0:00
CHMOD    ASM      1542  89-09-26   20:00
CHMOD    LST      6550  89-09-26   20:06
CHMOD    OBJ       398  89-09-26   20:06
        7 個のファイルがあります.
   937984 バイトが使用可能です.

A>
```

アセンブル終了後のディレクトリの状態を見る

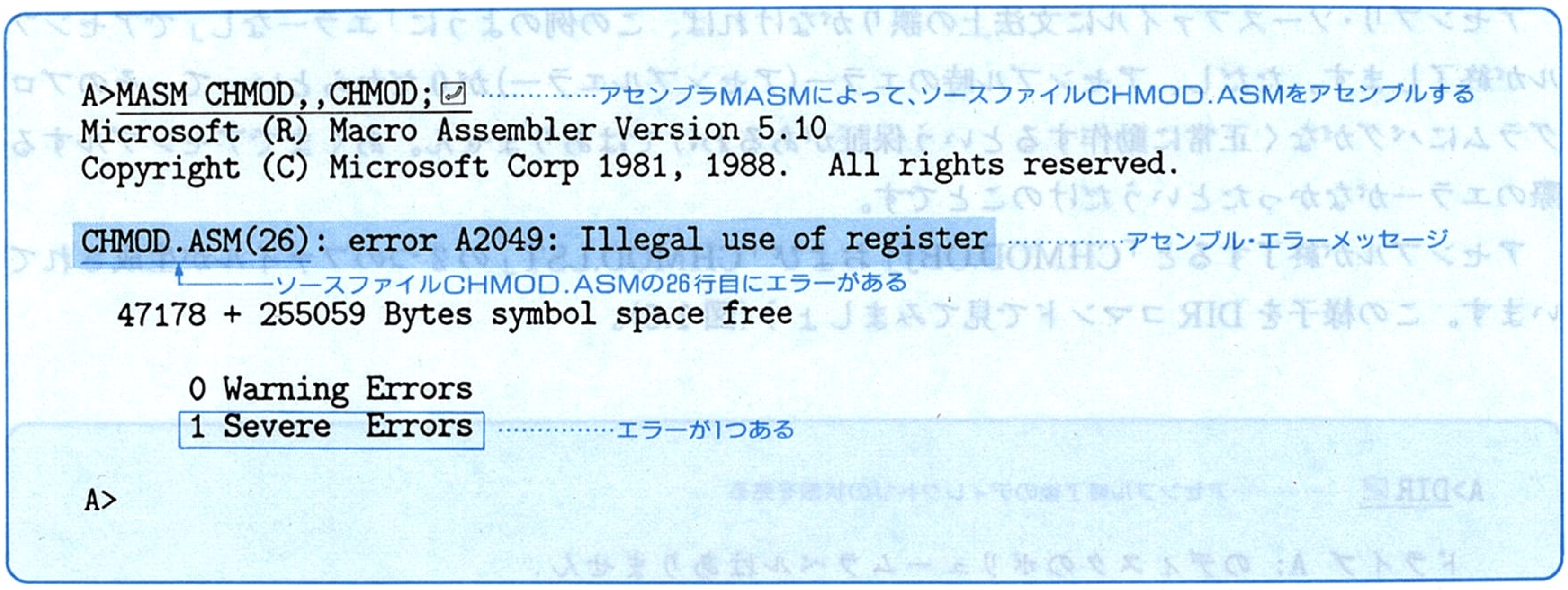
リスティングファイル
オブジェクトファイル
アセンブルによって生成された

図 1.6　アセンブルにより生成された各種のファイル

この「.OBJ」ファイルは、アセンブルの実行により生成されたマシン語形式のプログラムファイル（**オブジェクトファイル**）ですが、このOBJ形式のファイルは、まだ実行可能な状態ではありません。実行可能な最終的なプログラムを作成するには、このOBJファイルに対して、作業3以降の処理を行うことが必要です。

アセンブルによって生成されるもう1つのファイルCHMOD.LSTは、**リスティングファイル**と呼ばれるテキスト形式のファイルです。これは図1.8に示すように、入力されたもとのアセンブリ・ソースファイルの各行に、生成されたマシン語のコードが対応して書き込まれたファイルで、デバッグ時に役立つ補足的なファイルです。このファイルは、ソースファイルと同じテキスト形式ですので、TYPEコマンドで読むことができます。

さて、図1.5に示したアセンブルの実行例はアセンブルエラーがない場合の例でしたが、**図1.7**のようなエラーメッセージが出力された場合は、ソースファイルのどこかに誤りがあります。

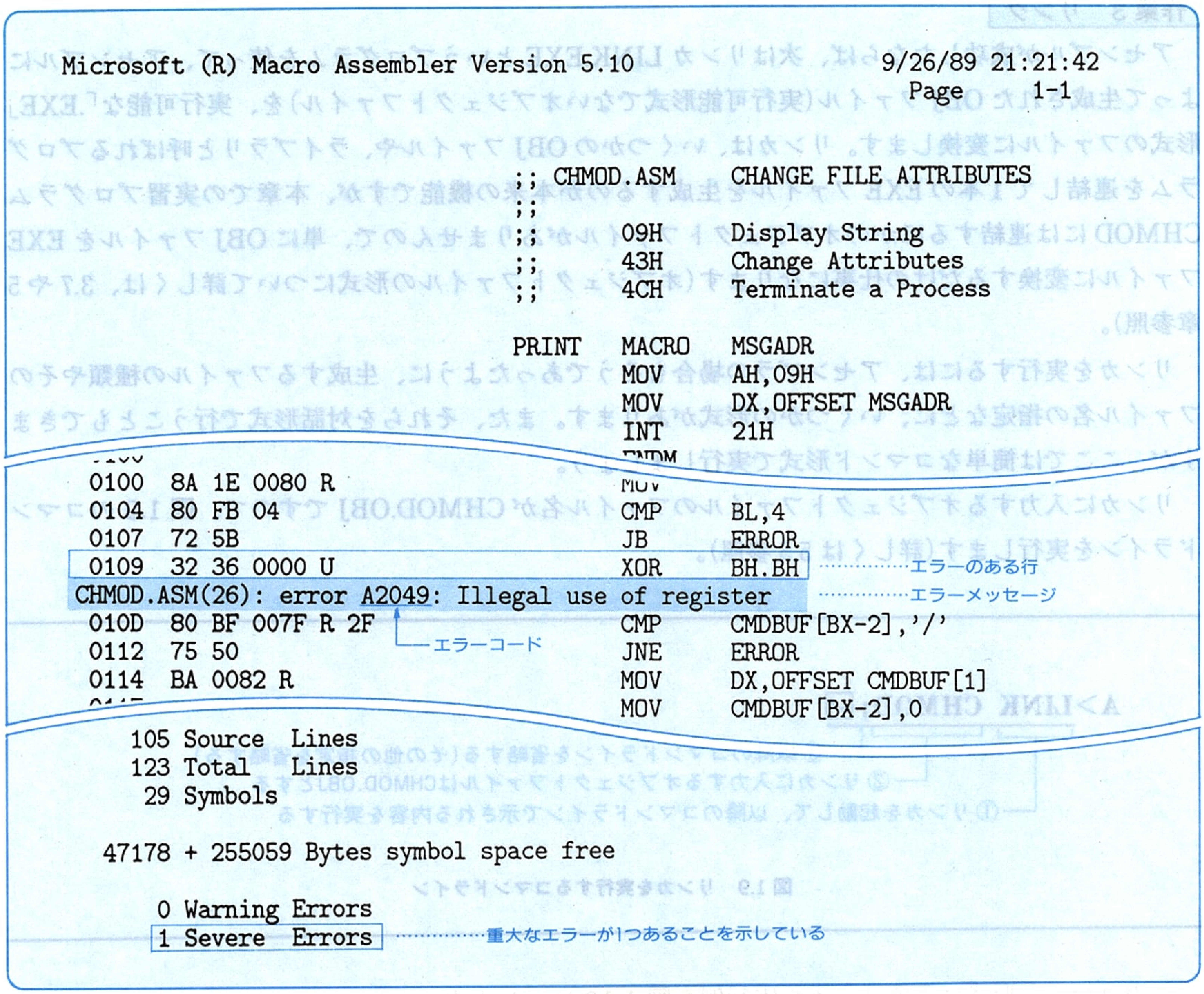


```
A>MASM CHMOD,,CHMOD;↵
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.

CHMOD.ASM(26): error A2049: Illegal use of register

  47178 + 255059 Bytes symbol space free

      0 Warning Errors
      1 Severe  Errors

A>
```

アセンブラMASMによって、ソースファイルCHMOD.ASMをアセンブルする
アセンブル・エラーメッセージ
ソースファイルCHMOD.ASMの26行目にエラーがある
エラーが1つある

図1.7　アセンブルエラーが発生したアセンブル実行例

ソースファイルに文法上の誤りがある場合、誤りのある行の行番号とエラーの種類が画面上に表示されます。また、リスティングファイルCHMOD.LSTにも誤りの箇所などが示されていますので、このファイルをTYPEコマンドで見れば、エラーの場所や種類がわかります。**図1.8**にその例を示しましょう。ただし、アセンブラのバージョンによっては、これらのエラー表示が異なる場合があります。



```
 Microsoft (R) Macro Assembler Version 5.10                9/26/89 21:21:42
                                                              Page     1-1


                                          ;; CHMOD.ASM     CHANGE FILE ATTRIBUTES
                                          ;;
                                          ;;       09H     Display String
                                          ;;       43H     Change Attributes
                                          ;;       4CH     Terminate a Process

                                          PRINT    MACRO   MSGADR
                                                   MOV     AH,09H
                                                   MOV     DX,OFFSET MSGADR
                                                   INT     21H

   0100  8A 1E 0080 R
   0104  80 FB 04                                  CMP     BL,4
   0107  72 5B                                     JB      ERROR
   0109  32 36 0000 U                              XOR     BH.BH
 CHMOD.ASM(26): error A2049: Illegal use of register
   010D  80 BF 007F R 2F                           CMP     CMDBUF[BX-2],'/'
   0112  75 50                                     JNE     ERROR
   0114  BA 0082 R                                 MOV     DX,OFFSET CMDBUF[1]
                                                   MOV     CMDBUF[BX-2],0

      105 Source  Lines
      123 Total   Lines
       29 Symbols

    47178 + 255059 Bytes symbol space free

        0 Warning Errors
        1 Severe  Errors
```

エラーのある行

エラーメッセージ

エラーコード

重大なエラーが1つあることを示している

図 1.8　リスティングファイルでアセンブルエラーの起きた箇所を見る

アセンブルエラーの発生したソースファイルを調べると、

XOR　BH.BH

の行がエラーとなっており、よく見るとカンマ「,」であるべきものがピリオド「.」になっています。エラー表示の「A2049」の意味は、アセンブラ(A)の重大なエラー(2)で、エラーの種類を識別するエラーコードが49番(049)ということであり、このコードの意味は、コメントとしても表示されているように「Illegal use of register」(レジスタの使い方が誤っている)というエラーメッセージです。アセンブルをした結果、このようなソースファイルの誤りがあればエディタを使って訂正し、再度アセンブルしてください。アセンブルエラーが発生せずにアセンブルが終了したならば、次のリンク作業を行います(作業3)。

作業３　リンク

アセンブルが成功したならば、次はリンカ LINK.EXE というプログラムを使って、アセンブルによって生成された OBJ ファイル(実行可能形式でないオブジェクトファイル)を、実行可能な「.EXE」形式のファイルに変換します。リンカは、いくつかの OBJ ファイルや、ライブラリと呼ばれるプログラムを連結して１本の EXE ファイルを生成するのが本来の機能ですが、本章での実習プログラム CHMOD には連結するほかのオブジェクトファイルがありませんので、単に OBJ ファイルを EXE ファイルに変換するだけの仕事になります(オブジェクトファイルの形式について詳しくは、3.7 や 5 章参照)。

リンカを実行するには、アセンブラの場合もそうであったように、生成するファイルの種類やそのファイル名の指定などに、いくつかの形式があります。また、それらを対話形式で行うこともできますが、ここでは簡単なコマンド形式で実行しましょう。

リンカに入力するオブジェクトファイルのファイル名が CHMOD.OBJ ですので、**図 1.9** のコマンドラインを実行します(詳しくは 5.5 参照)。

```
A>LINK CHMOD;⏎
```

③ 以降のコマンドラインを省略する(その他の指定を省略する)

② リンカに入力するオブジェクトファイルはCHMOD.OBJとする

① リンカを起動して、以降のコマンドラインで示される内容を実行する

図 1.9　リンカを実行するコマンドライン

このコマンド形式によるリンカの実行例を図 1.10 に示します。

```
A>LINK CHMOD;⏎

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

LINK : warning L4021: no stack segment

A>
```

ファイルCHMOD.OBJに対してリンカLINKを実行する

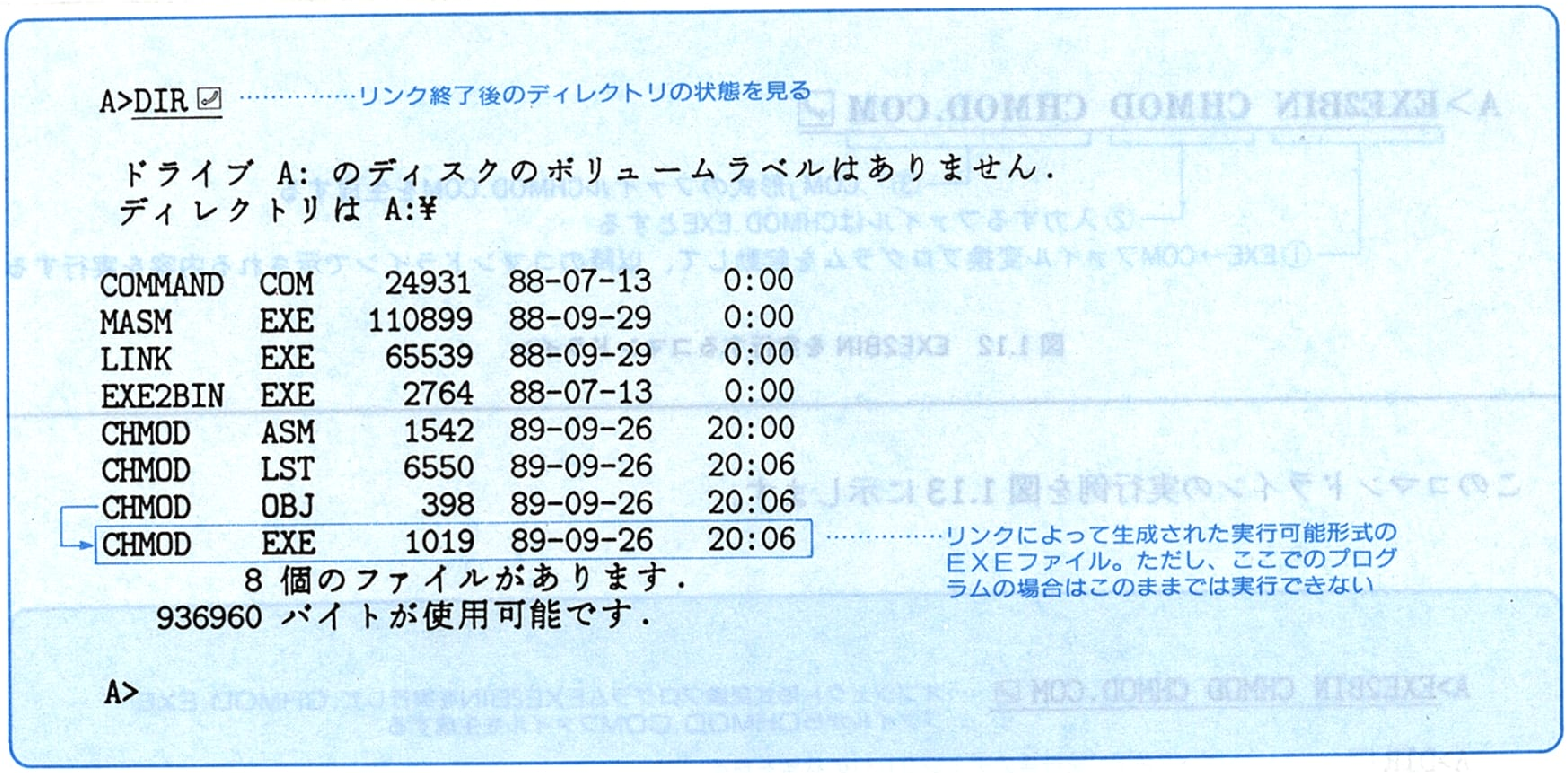
このエラーメッセージは、ここでのプログラムの場合は無視してよい

リンカの実行終了

図 1.10　リンカの実行例

リンカの実行が終了しましたが、「no stack segment」のエラーが発生したという警告が表示されています(ただし、LINK.EXEのバージョンによっては、このエラー表示が異なる場合がある)。これは、ソースファイルを簡単にするために、このプログラムが動作する際に内部的に使用するスタックと呼ばれるメモリ領域を用意しなかったためですが、ここでは気にする必要はありません。

このリンカの実行によって、EXEファイルが生成されていますので、DIRコマンドで確認してみましょう(図1.11)。



```
A>DIR

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13    0:00
MASM     EXE    110899  88-09-29    0:00
LINK     EXE     65539  88-09-29    0:00
EXE2BIN  EXE      2764  88-07-13    0:00
CHMOD    ASM      1542  89-09-26   20:00
CHMOD    LST      6550  89-09-26   20:06
CHMOD    OBJ       398  89-09-26   20:06
CHMOD    EXE      1019  89-09-26   20:06
         8 個のファイルがあります.
    936960 バイトが使用可能です.

A>
```

リンク終了後のディレクトリの状態を見る

リンクによって生成された実行可能形式のEXEファイル。ただし、ここでのプログラムの場合はこのままでは実行できない

図1.11　リンカの実行により生成されたEXEファイル

リンカにより生成されたEXEファイルは、本来は実行可能なオブジェクトファイルですが、このCHMODプログラムは、自分自身でスタックの処理などを行っていないなどのため、このままでは実行できません。実行可能なオブジェクトファイルを得るには、さらに次の作業を行って、このCHMOD.EXEプログラムを、COMファイルに変換します。

作業4　オブジェクト形式の変換「.EXE」→「.COM」

実行可能なCHMODプログラムを作るには、CHMOD.EXEから、もう1つの実行可能形式である「.COM」形式のファイルを生成します。COMファイルの場合には、実行時にMS-DOSによってスタックが用意されることなどによりプログラムの実行が可能になるのです。この「.EXE」から「.COM」への変換には、オブジェクト形式変換プログラム**EXE2BIN.EXE**を実行します。このプログラムは通常、図**1.12**のコマンドラインにより実行します。

```
A>EXE2BIN CHMOD CHMOD.COM⏎
```

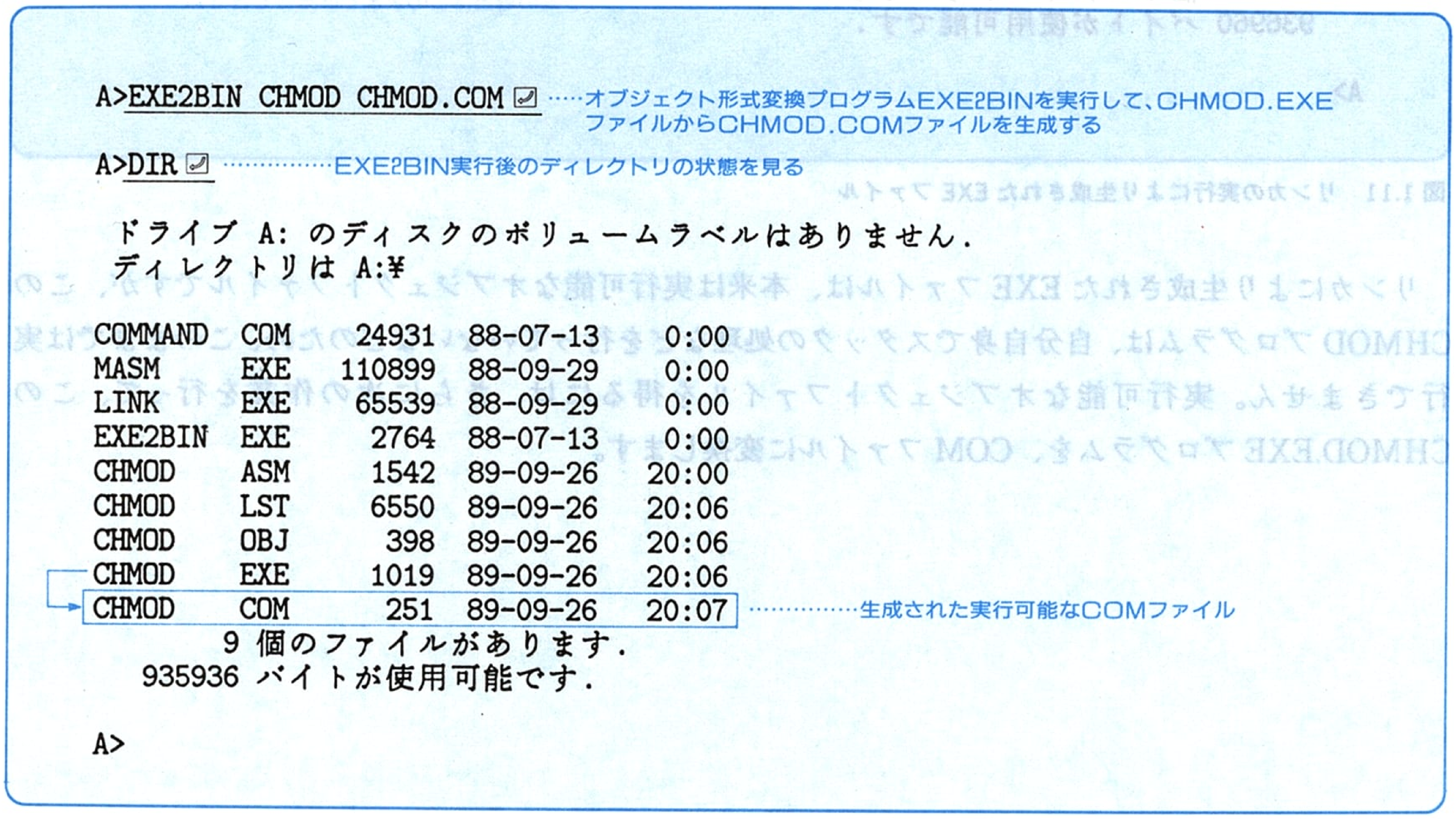
③「.COM」形式のファイルCHMOD.COMを生成する

②入力するファイルはCHMOD.EXEとする

①EXE→COMファイル変換プログラムを起動して、以降のコマンドラインで示される内容を実行する

図1.12　EXE2BINを実行するコマンドライン

このコマンドラインの実行例を図**1.13**に示します。



```
A>EXE2BIN CHMOD CHMOD.COM⏎
A>DIR⏎

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM    24931  88-07-13   0:00
MASM     EXE   110899  88-09-29   0:00
LINK     EXE    65539  88-09-29   0:00
EXE2BIN  EXE     2764  88-07-13   0:00
CHMOD    ASM     1542  89-09-26  20:00
CHMOD    LST     6550  89-09-26  20:06
CHMOD    OBJ      398  89-09-26  20:06
CHMOD    EXE     1019  89-09-26  20:06
CHMOD    COM      251  89-09-26  20:07
         9 個のファイルがあります.
   935936 バイトが使用可能です.

A>
```

……オブジェクト形式変換プログラムEXE2BINを実行して、CHMOD.EXEファイルからCHMOD.COMファイルを生成する
……EXE2BIN実行後のディレクトリの状態を見る
……生成された実行可能なCOMファイル

図1.13　「.EXE」→「.COM」ファイル変換プログラムの実行

EXE2BINの実行により、EXEファイルから、もう1つの実行可能形式のファイルCHMOD.COMが生成されました。このCOMファイルは、ここでのプログラムのような、スタックの自己処理を行っていないプログラムでも実行可能です(この違いは重要であり、詳しくは5章で解説する)。

以上の作業で、ユーザーが最終的に利用できる形式のCHMODプログラムが完成しました。ただし、実際のソフトウェア開発では、でき上がった最初のプログラムが1回で完動することはまずあり得ませんので、このあとに**デバッグ作業**を行うことになります。普通この作業はかなりたいへんで、開発過程全体の大きなウエイトを占めることになります。

1.4　CHMODプログラムの実行と応用

作成したCHMODプログラムは、1.2節でそのコマンド形式と機能を解説したとおり、次の2種類のファイル属性を設定することができます。

●任意のファイルを隠しファイルにする／しない
●任意のファイルをリードオンリーファイルにする／しない

ただし、すでにシステム属性が付けられていて、システムファイルに設定されているファイル(たとえばIO.SYS、MSDOS.SYSなど)は、通常のファイルにすることはできません。たとえば、システムファイルに対して、リード／ライト可能な属性を設定したとしても、システム属性が設定されているため、DELコマンドは無効になります。

CHMODプログラムは、たとえば他人に存在を知られたくないファイルを隠しファイルにしたり(見ようとすれば外部プログラムのCHKDSK.EXE(または.COM)で見られるが、内蔵コマンドのDIRコマンドでは見られない)、誤操作などによって消されたり書き換えられては困る重要なファイルを、消去／書き換えができないファイルにして保護するときなどに利用します。とくに大勢の人が共用するフロッピーディスクや、非常に多くのファイルが混在する大容量のハードディスクのファイル管理には、このCHMODプログラムの活用をお勧めします。

では、作成したCHMODプログラムの代表的な実行例をいくつか示しましょう(**図1.14**)。

```
A>DIR↵ ……………CHMODプログラム実行前のディレクトリの状態を見る

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13    0:00
MASM     EXE    110899  88-09-29    0:00
LINK     EXE     65539  88-09-29    0:00
EXE2BIN  EXE      2764  88-07-13    0:00
CHMOD    ASM      1542  89-09-26   20:00 ……………このファイル(CHMODのソースファイル)を対象に、
                                                   ファイル属性を操作してみよう
CHMOD    LST      6550  89-09-26   20:06
CHMOD    OBJ       398  89-09-26   20:06
CHMOD    EXE      1019  89-09-26   20:06
CHMOD    COM       251  89-09-26   20:07
         9 個のファイルがあります.
    935936 バイトが使用可能です.

A>CHMOD CHMOD.ASM/?↵ ……………CHMODプログラムを/?スイッチを付けて実行し、
                              ファイルCHMOD.ASMの属性を調べる

Read/Write File ……………リード／ライト可能な通常のファイルである

A>CHMOD CHMOD.ASM/H↵ ……………CHMODプログラムを/Hスイッチを付けて実行し、
                              ファイルCHMOD.ASMを隠しファイルにする

A>DIR↵ ……………ディレクトリの状態を見る

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13    0:00
MASM     EXE    110899  88-09-29    0:00
LINK     EXE     65539  88-09-29    0:00
EXE2BIN  EXE      2764  88-07-13    0:00
CHMOD    LST      6550  89-09-26   20:06 ←――CHMOD.ASMが見えなくなった
CHMOD    OBJ       398  89-09-26   20:06
CHMOD    EXE      1019  89-09-26   20:06
CHMOD    COM       251  89-09-26   20:07
         8 個のファイルがあります.
    935936 バイトが使用可能です.

A>CHMOD CHMOD.ASM/?↵ ……………/?スイッチでファイルCHMOD.ASMの属性を調べる

Read/Write Hidden File ……………リード／ライト可能で、かつ隠しファイルである

A>CHMOD CHMOD.ASM/N↵ ……………/Nスイッチでファイルを CHMOD.ASMを見えるファイルにする

A>DIR↵ ……………ディレクトリの状態を調べる

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥
```

― 図 1.14 ― (次ページに続く)

```
COMMAND  COM    24931  88-07-13   0:00
MASM     EXE   110899  88-09-29   0:00
LINK     EXE    65539  88-09-29   0:00
EXE2BIN  EXE     2764  88-07-13   0:00
CHMOD    ASM     1542  89-09-26  20:00 ……………もとどおり見えるようになった
CHMOD    LST     6550  89-09-26  20:06
CHMOD    OBJ      398  89-09-26  20:06
CHMOD    EXE     1019  89-09-26  20:06
CHMOD    COM      251  89-09-26  20:07
        9 個のファイルがあります.
   935936 バイトが使用可能です.

A>CHMOD CHMOD.ASM/R⏎ ……………/RスイッチでファイルCHMOD.ASMをリードオンリーファイルにする

A>DEL CHMOD.ASM⏎ ……………DELコマンドでCHMOD.ASMを消去しようとする
アクセスは拒否されました. ……………このようなエラーメッセージが表示され、DELコマンドは実行されない

A>DIR⏎ ……………ディレクトリの状態を見る

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM    24931  88-07-13   0:00
MASM     EXE   110899  88-09-29   0:00
LINK     EXE    65539  88-09-29   0:00
EXE2BIN  EXE     2764  88-07-13   0:00
CHMOD    ASM     1542  89-09-26  20:00 ……………DELコマンドを実行したが消去されていない
CHMOD    LST     6550  89-09-26  20:06
CHMOD    OBJ      398  89-09-26  20:06
CHMOD    EXE     1019  89-09-26  20:06
CHMOD    COM      251  89-09-26  20:07
        9 個のファイルがあります.
   935936 バイトが使用可能です.

A>CHMOD CHMOD.ASM/?⏎ ……………/?スイッチでファイルCHMOD.ASMの属性を調べる

Read Only File ……………リードオンリーファイルになっている

A>
```

図 1.14　CHMOD プログラムの実行例

1.5　本章はMS-DOS解説のためのプロローグ

CHMODプログラムの作成実習はいかがでしたか？　ここではまだそれぞれの意味がわからなくても、本書を読み進むうちに、それらの疑問が次々と明らかになっていくでしょう。

MS-DOSを開発レベルで利用するには、最低でも本書で取り上げられている程度の知識が必要であり、MS-DOS全体を理解するとなると、さらに広く深い知識が必要となります。本章で行ったCHMODプログラムの作成実習で、私たちはMS-DOSの世界をかいま見たわけですが、これをきっかけにして、2章からはMS-DOSの内部を探っていきましょう。

その前に、CHMODプログラムがMS-DOSのどの部分の知識をもとに設計され、作成されているかについて、ざっと触れておきます。

ファイル属性についての知識

ディスクファイルの性質を決定する**ファイル属性**についての知識。「2章 MS-DOSのファイルシステム」で解説する。

システムコールについての知識

MS-DOSシステムの各種の機能（ファンクション）を、ユーザープログラムによって呼び出し、利用する知識。CHMODプログラムでは、文字の入出力や、ファイル属性を設定するファンクションなどを利用している。「3章 MS-DOSの仕組みと働き」では理論的に、「4章　システムコールとソフトウェア割り込み」では具体的に解説する。

アセンブラおよびリンカについての知識

アセンブラによるソフトウェア開発の一般的な知識のほか、MS-DOSの標準アセンブラ（MASM）およびリンカ（LINK）の使い方の知識。「5章　アセンブラによるソフトウェア開発」で解説する。

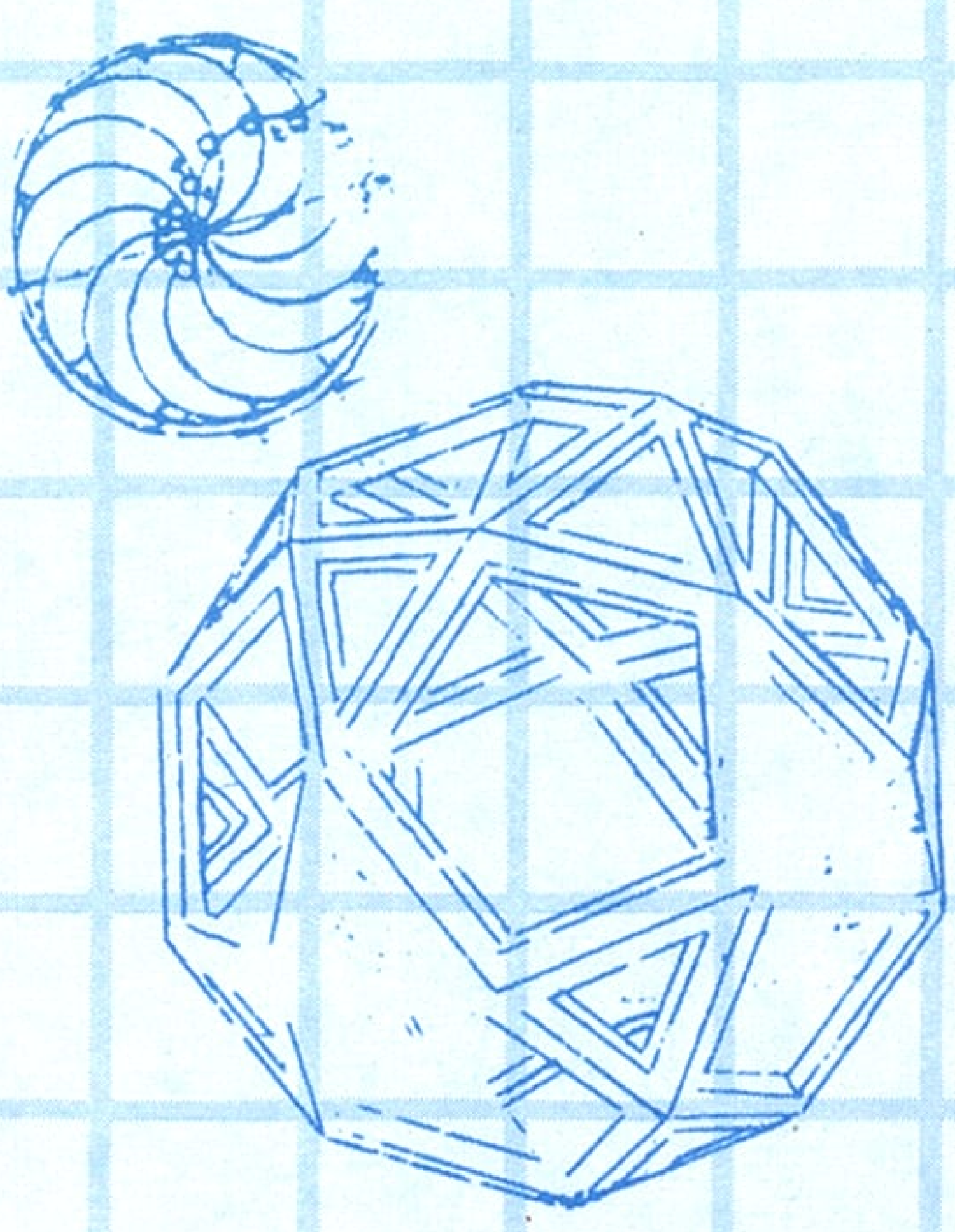
実行可能な2つのオブジェクトファイル（「.EXE」と「.COM」）についての知識

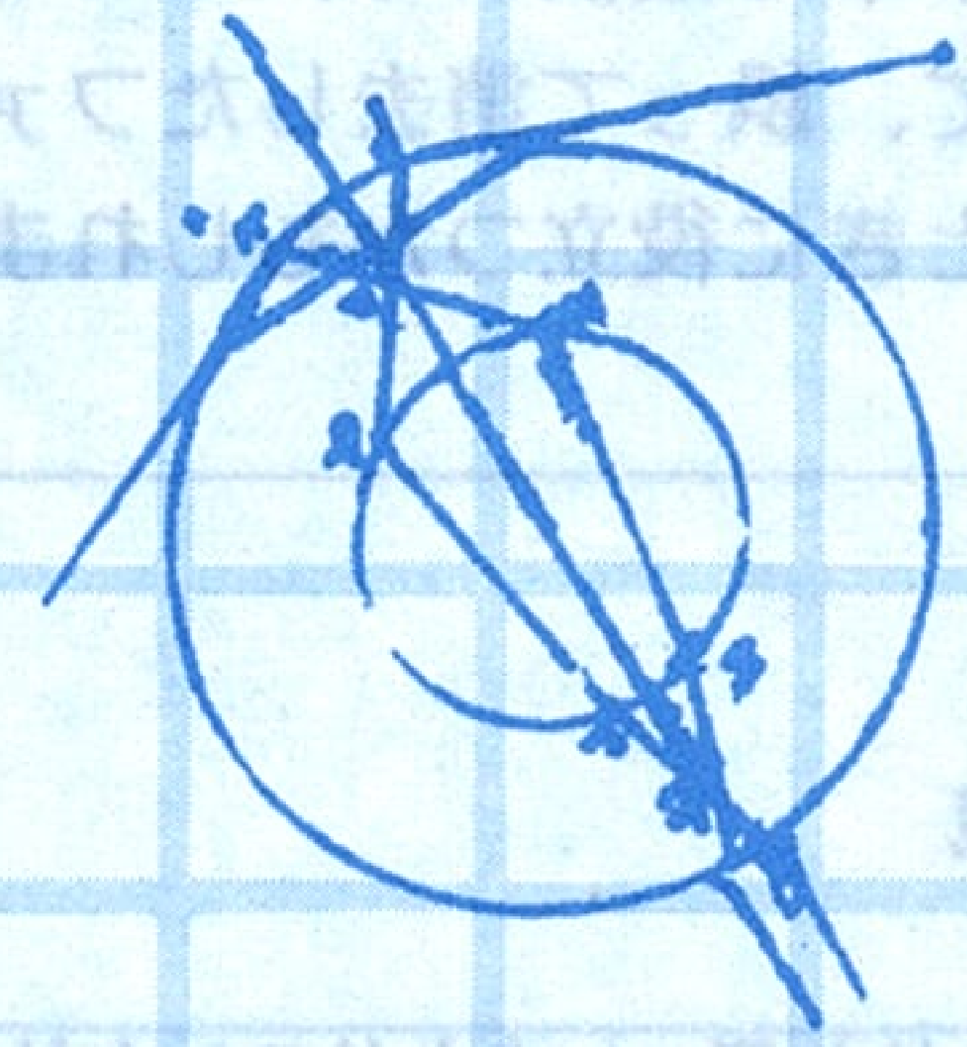
MS-DOS上でプログラムを実行する際の**メモリ管理**、**プロセス管理**に関連し、ソフトウェア開発の知識にも関連する。「3章　MS-DOSの内部構造」では理論的に、「5章 アセンブラによるソフトウェア開発」で具体的に解説する。

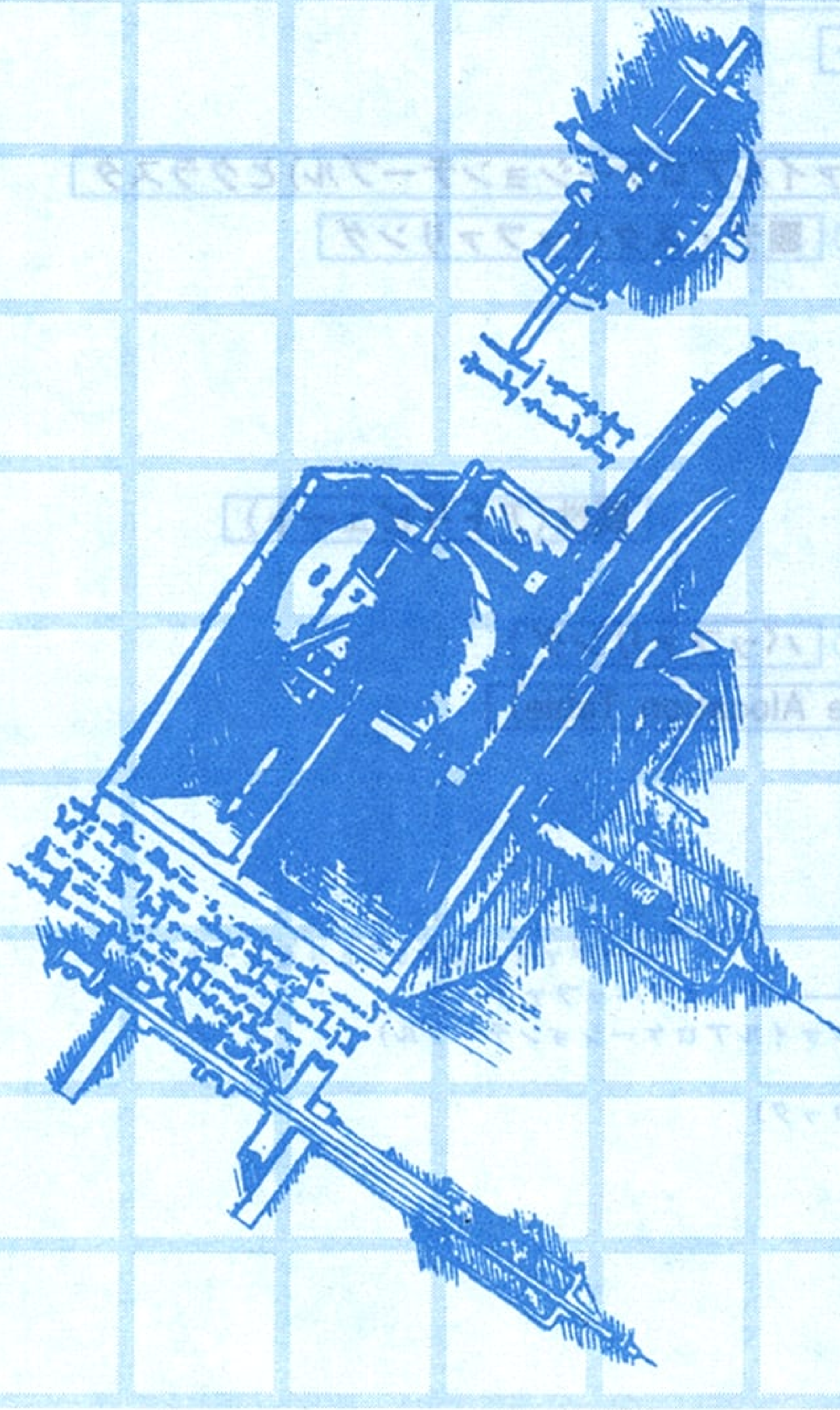
ざっとこれらの項目が、CHMODプログラムの作成にとくに関係する知識ですが、これら相互の関連も重要です。MS-DOSを応用するには、まず土台となるMS-DOSシステム全般の基礎知識の上にMS-DOSを利用するための具体的な知識が必要なのです。これには、1つひとつを順に理解していくほかにないでしょう。

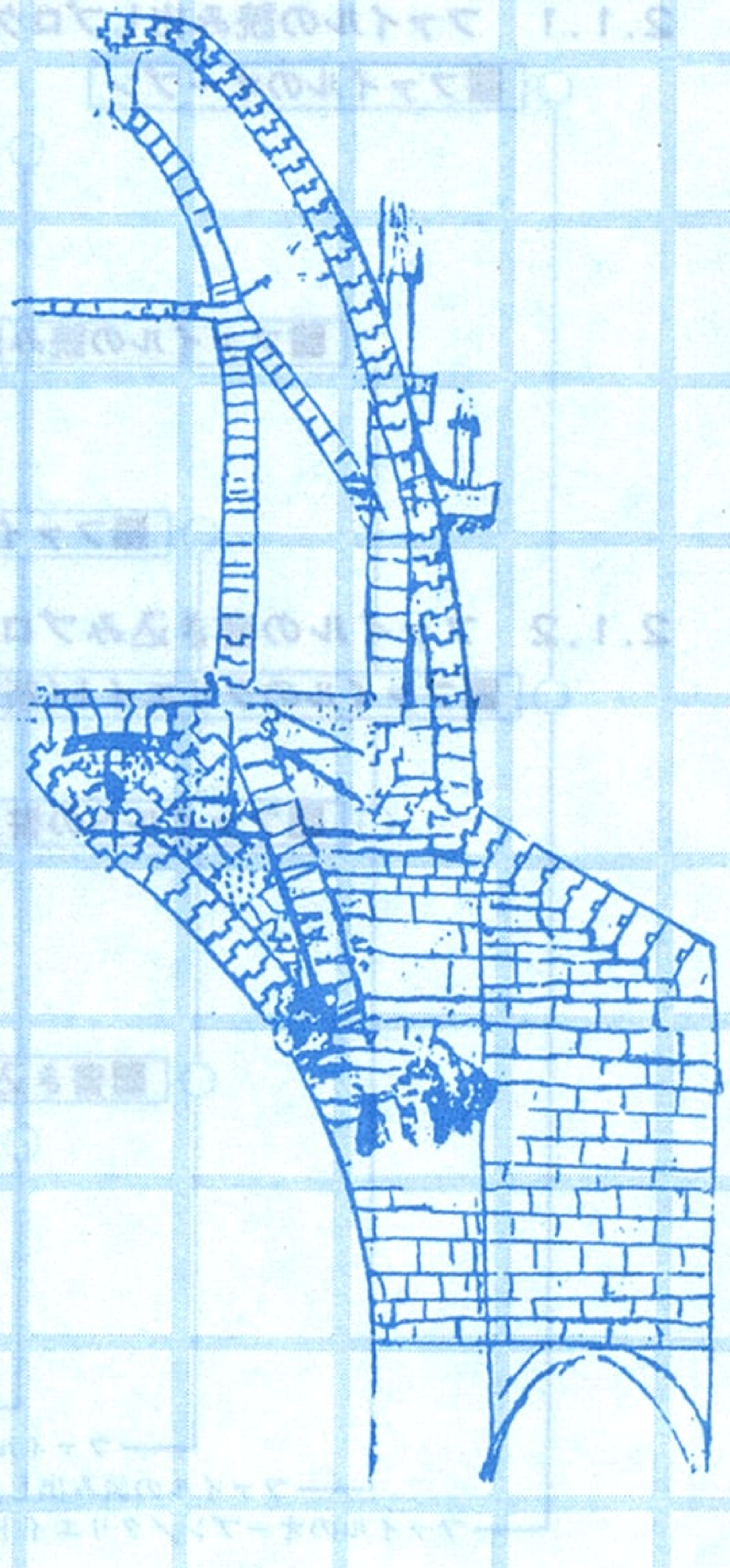
では本章で行ったソフトウェア開発実習をきっかけとして、次章の『MS-DOSのファイルシステム』から順に手を付けていきましょう。

2章
MS-DOSのファイルシステム







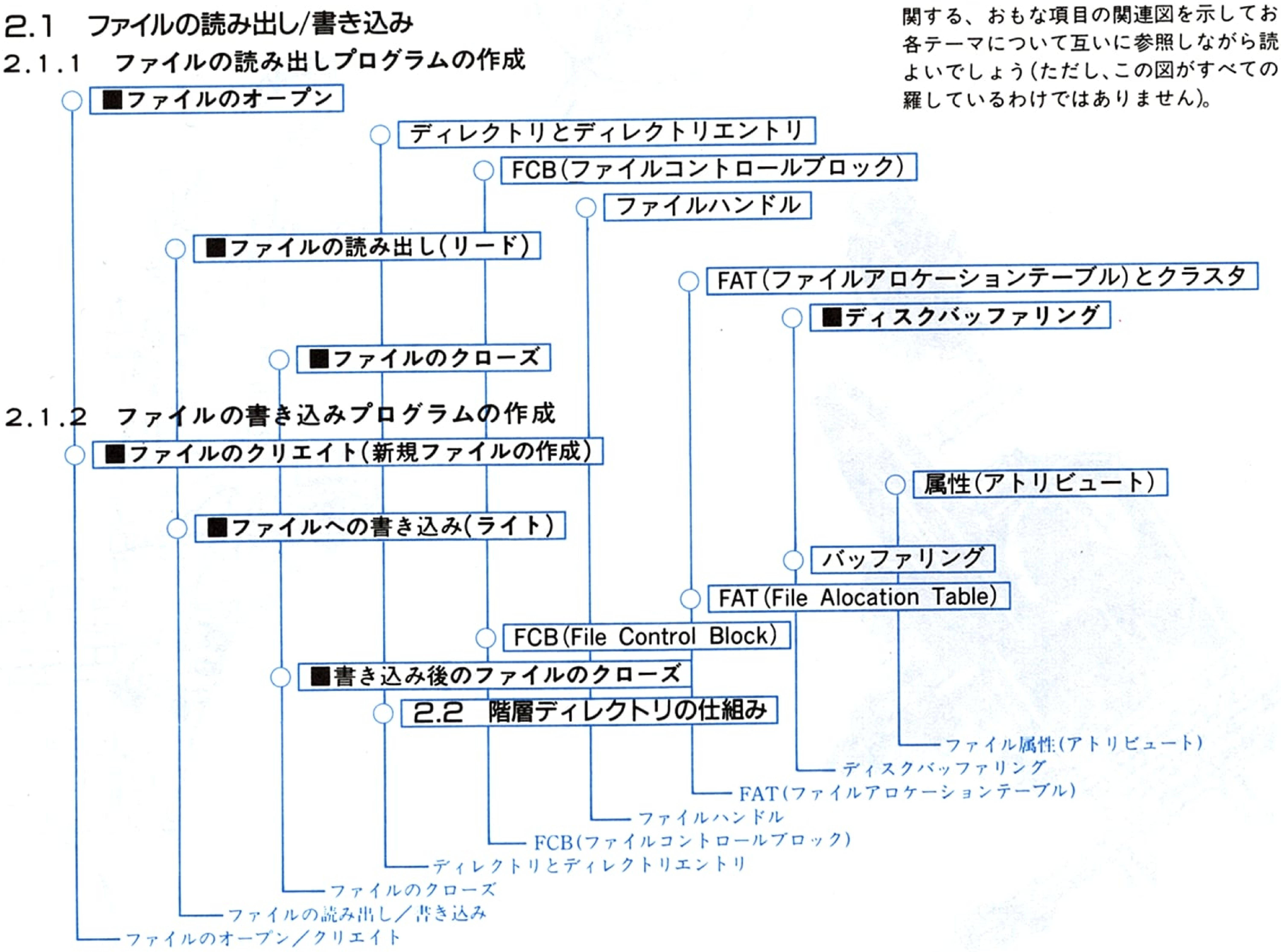


OS（オペレーティングシステム）とは何か、それはどのような働きをするのかなどの理論的な解説は3章で行いますが、MS-DOSの多くの機能の中でも、その中心はなんといってもファイルの管理です。このファイルを管理するメカニズムを「ファイルシステム」と呼んでいます。MS-DOSでは、コンソールの入出力なども「ファイル」という概念で取り扱いますが、そのような論理的なファイルについては3章で解説することにして、本章ではディスク上の実際のファイルの管理について解説しましょう。

MS-DOSのファイルシステムは、ディスク上にデータを書き込んだり、それを読み出したり、更新したり、削除したり、さらにこれらの操作を階層ディレクトリ上でサポートしたりしなければなりません。それも高速で処理することが要求されます。

このような管理がどのような仕組みで行われているのか、本章ではこれらを解き明かしていきましょう。また、本章の最後には、ここで解説したファイルシステムの知識をフルに活用して、誤って消去したファイルを復活する操作を実例で示します。もしものときに役立つかもしれません。

本章では、各テーマが散在していますので、2.1に関する、おもな項目の関連図を示しておきます。各テーマについて互いに参照しながら読み進むとよいでしょう（ただし、この図がすべての関連を網羅しているわけではありません）。


2.1　ファイルの読み出し/書き込み
2.1.1　ファイルの読み出しプログラムの作成
■ファイルのオープン
ディレクトリとディレクトリエントリ
FCB（ファイルコントロールブロック）
ファイルハンドル
■ファイルの読み出し（リード）
FAT（ファイルアロケーションテーブル）とクラスタ
■ディスクバッファリング
■ファイルのクローズ
2.1.2　ファイルの書き込みプログラムの作成
■ファイルのクリエイト（新規ファイルの作成）
属性（アトリビュート）
■ファイルへの書き込み（ライト）
バッファリング
FAT（File Alocation Table）
FCB（File Control Block）
■書き込み後のファイルのクローズ
2.2　階層ディレクトリの仕組み
ファイル属性（アトリビュート）
ディスクバッファリング
FAT（ファイルアロケーションテーブル）
ファイルハンドル
FCB（ファイルコントロールブロック）
ディレクトリとディレクトリエントリ
ファイルのクローズ
ファイルの読み出し／書き込み
ファイルのオープン／クリエイト

2.1　ファイルの読み出し／書き込み

ファイルシステムの中心は、ディスク上のファイルの管理です。では MS-DOS のファイルシステムは、どのような仕組みでそれを管理し、またアクセスしているのでしょうか。これらを解説するために、本節ではディスク上のファイルを読み出したり、メモリ上のデータをディスクに書き込んだりする簡単なサンプルプログラムを実際に作成し、そこから MS-DOS のファイルシステムを解説していきましょう。

2.1.1　ファイルの読み出しプログラムの作成

ディスク上のファイルは、どのようなプログラムを作れば読み出すことができるのでしょうか。まず、ファイル読み出しの最も簡単なサンプルプログラム（プログラム名「FREAD」）を作成します。このプログラムは、MS-DOS の TYPE コマンドに相当するもので、テキストファイルを読み出してコンソール（ディスプレイ）に出力します。このプログラムのアセンブリ・ソースプログラムを**リスト 2.1** に示します。

```
;; FREAD.ASM     File Read Program
;;
;;       02H     Display Character
;;       09H     Display String
;;       3DH     Open a File
;;       3EH     Close a File Handle
;;       3FH     Read From File/Device
;;       4CH     Terminate a Process

CSEG     SEGMENT
         ASSUME CS:CSEG,DS:CSEG,ES:CSEG
         ORG     80H
CMDLEN   DB      ?
CMDBUF   DB      127 DUP (?)

         ORG     100H
START:   MOV     BL,CMDLEN
         CMP     BL,2
         JB      NERROR
         XOR     BH,BH
         MOV     CMDBUF[BX],0
         MOV     DX,OFFSET CMDBUF[1]
```

＊このプログラムのさらに詳しい解説は、4章のリスト4.3で行っていますので、ここではその概要を示します。

入力されたコマンドラインのチェックと、その文字列の前処理

— リスト 2.1 — (次ページに続く)

```
        MOV     AX,3D00H  ……………指定されたファイルをオープンするファンクションリクエスト
        INT     21H
        JC      NERROR …………オープンできない場合(指定されたファイルがないなど)は、エラーメッセージの表示へ
        MOV     BX,AX

LOP:    MOV     AH,3FH
        MOV     DX,OFFSET BUF ……………ファイルの読み出し(リード)のファンクションリクエスト
        MOV     CX,1
        INT     21H
        TEST    AX,AX    ┐
        JZ      RDEND    │
        MOV     DL,BUF   │ 読み出したデータが、ファイルの最後であるかどうかをチェックする。
        CMP     DL,1AH   │ 最後であればプログラムの終了処理へ
        JE      RDEND    ┘
        MOV     AH,02H  ……………読み出したデータをディスプレイに表示するファンクションリクエスト
        INT     21H
        JMP     SHORT LOP ……………これの繰り返し

RDEND:  MOV     AH,3EH  ……………ファイルをクローズするファンクションリクエスト
        INT     21H
        XOR     AL,AL ……………正常終了のコードを設定する
        JMP     SHORT RETURN

NERROR: MOV     DX,OFFSET NERRMES
        MOV     AH,09H          ……………文字列出力のファンクションリクエスト
        INT     21H                     エラーメッセージをディスプレイに表示する
        MOV     AL,1 ……………エラーコードを設定する
RETURN: MOV     AH,4CH  ……………プログラムの終了処理
        INT     21H

BUF     DB      0
NERRMES DB      0DH,0AH,"Not found",0DH,0AH,'$' ……………エラーメッセージ文字列
CSEG    ENDS
        END     START
```

リスト 2.1　ファイル読み出しプログラムのソースプログラム FREAD.ASM

このFREADプログラムの構成を**図 2.1**に、リスト 2.1のアセンブリ・ソースプログラムから、実行可能なオブジェクトプログラムFREAD.COMを作成する実行例を**図 2.2**に示します。

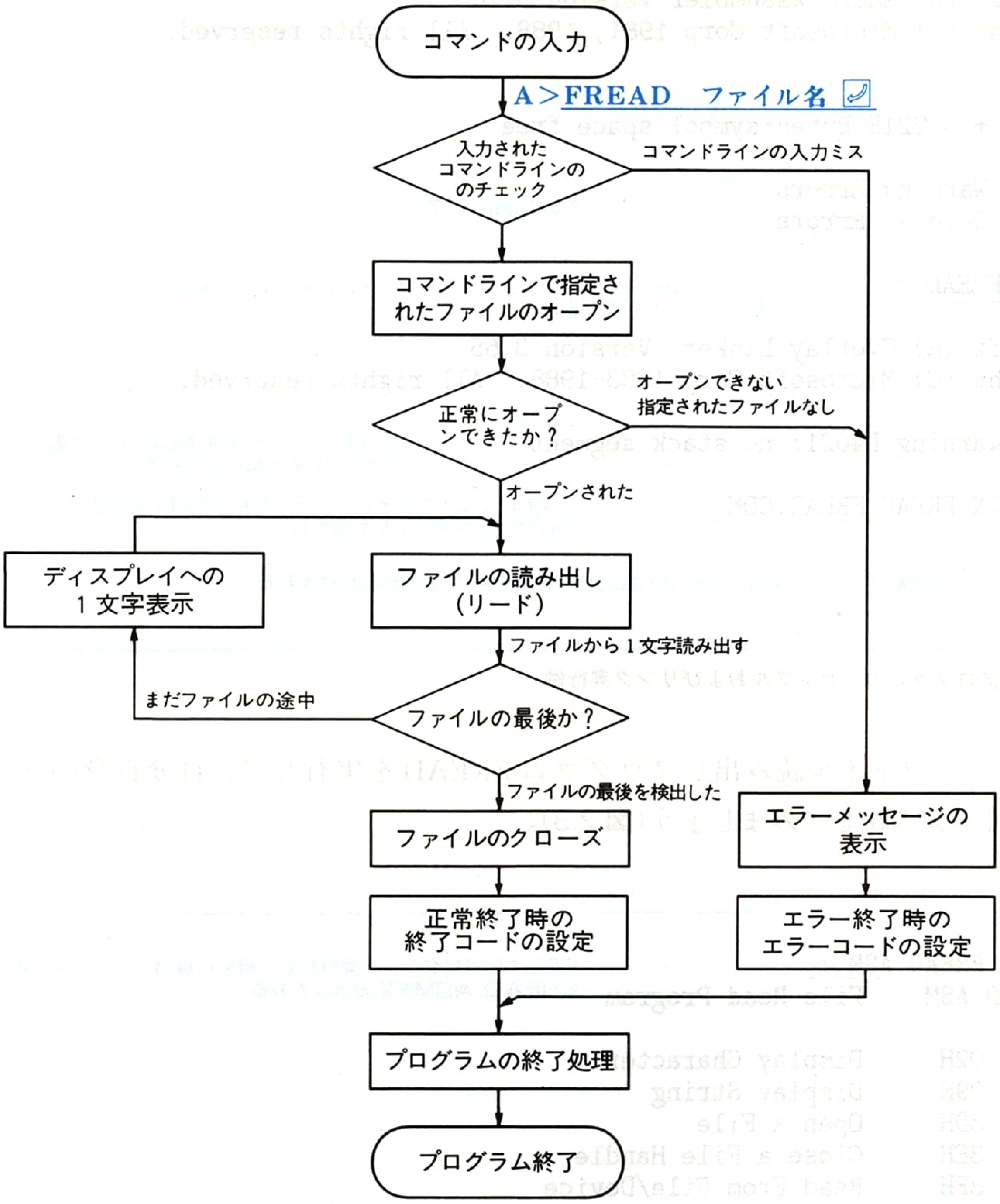
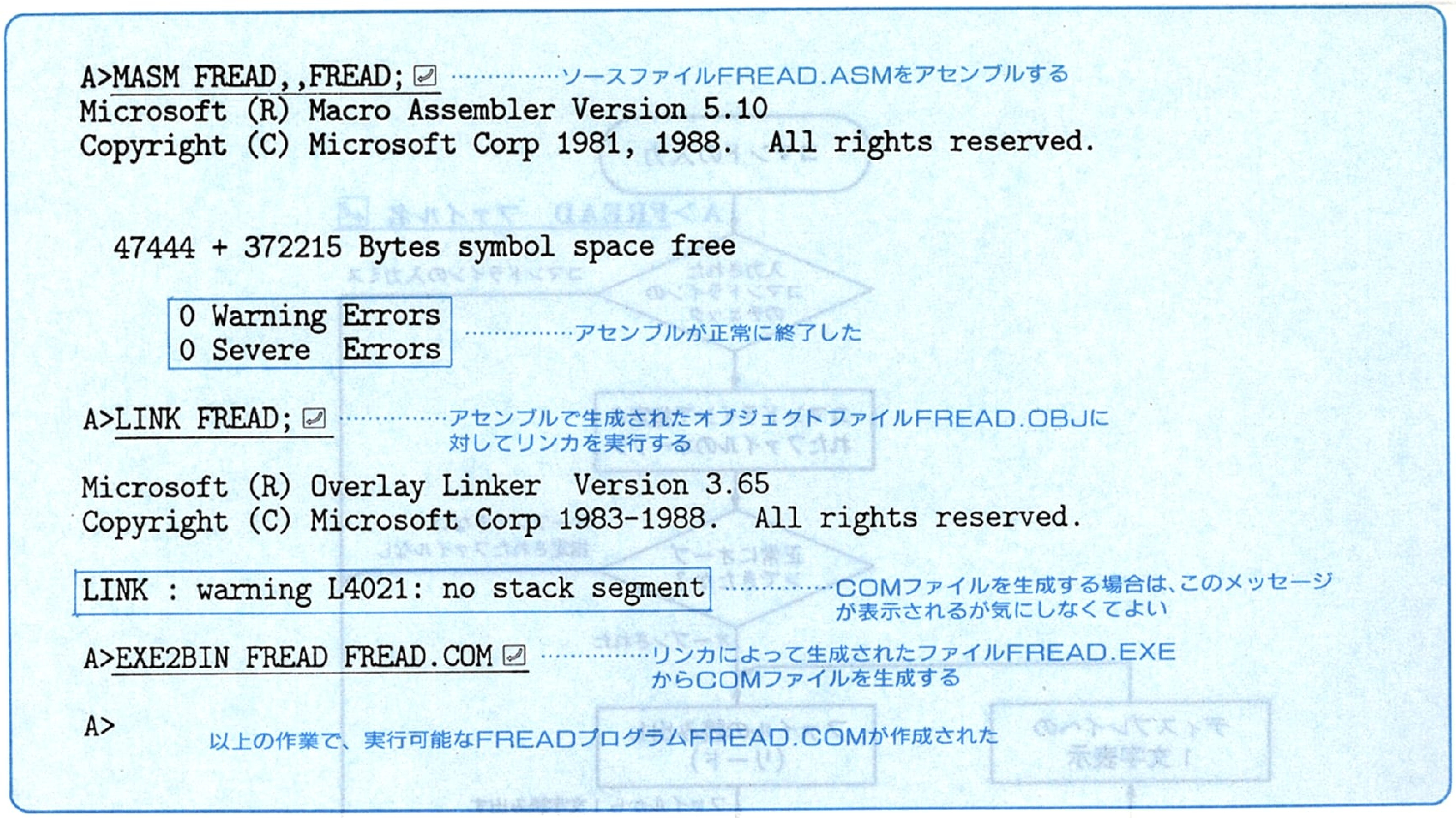

コマンドの入力
A＞FREAD　ファイル名
入力されたコマンドラインのチェック
コマンドラインの入力ミス
コマンドラインで指定されたファイルのオープン
正常にオープンできたか？
オープンできない
指定されたファイルなし
オープンされた
ディスプレイへの１文字表示
ファイルの読み出し（リード）
ファイルから１文字読み出す
まだファイルの途中
ファイルの最後か？
ファイルの最後を検出した
ファイルのクローズ
エラーメッセージの表示
正常終了時の終了コードの設定
エラー終了時のエラーコードの設定
プログラムの終了処理
プログラム終了

図 2.1　FREAD プログラムの構成



```
A>MASM FREAD,,FREAD;⏎
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


  47444 + 372215 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

A>LINK FREAD;⏎

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

LINK : warning L4021: no stack segment

A>EXE2BIN FREAD FREAD.COM⏎

A>
```

ソースファイルFREAD.ASMをアセンブルする

アセンブルが正常に終了した

アセンブルで生成されたオブジェクトファイルFREAD.OBJに対してリンカを実行する

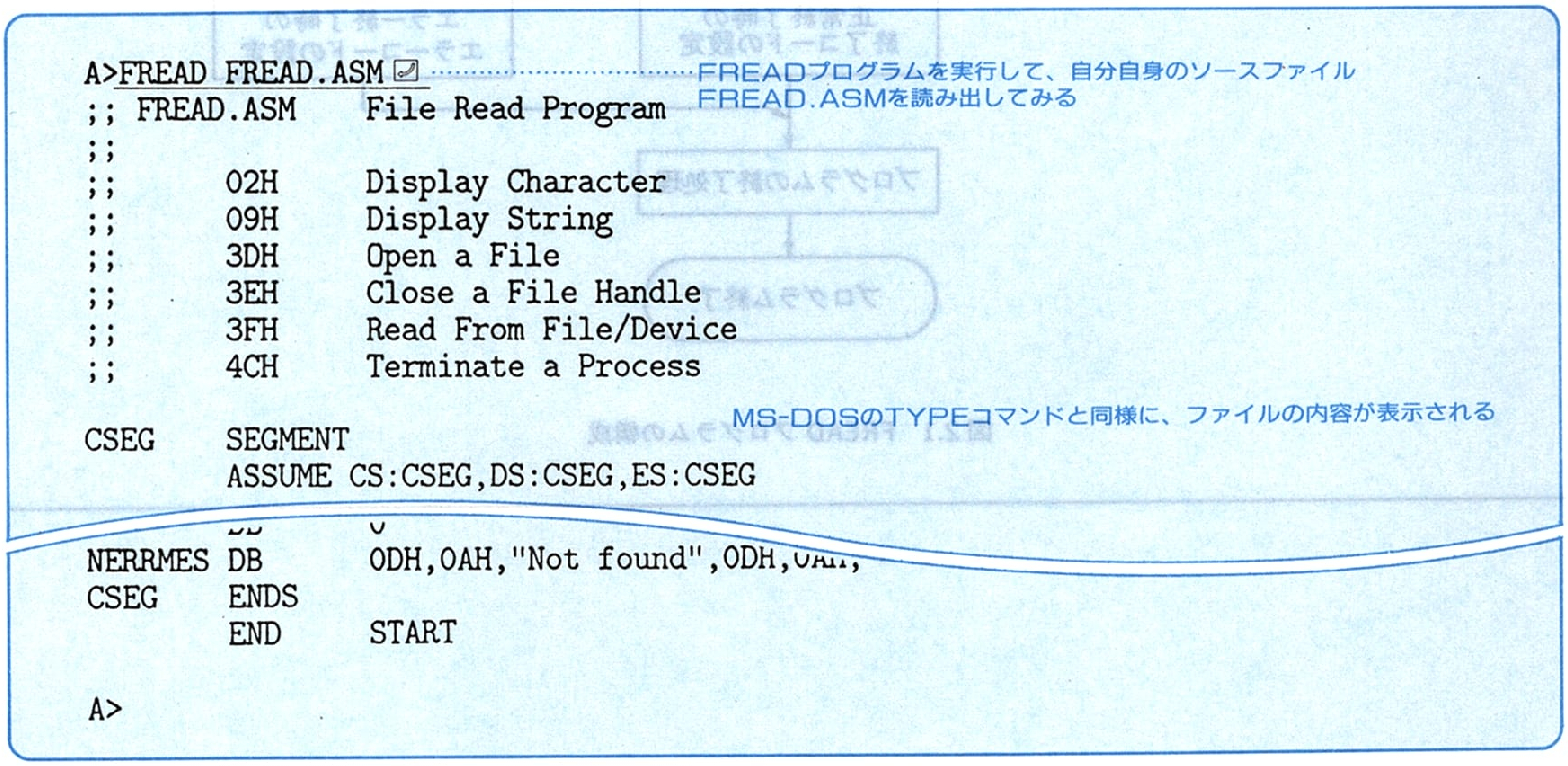
COMファイルを生成する場合は、このメッセージが表示されるが気にしなくてよい

リンカによって生成されたファイルFREAD.EXEからCOMファイルを生成する

以上の作業で、実行可能なFREADプログラムFREAD.COMが作成された

図2.2　FREADプログラムのアセンブルおよびリンク実行例

では、完成したファイル読み出しプログラムFREADを実行して、自分自身のソースファイルFREAD.ASMを読み出してみましょう（図2.3）。



```
A>FREAD FREAD.ASM⏎
;; FREAD.ASM     File Read Program
;;
;;       02H     Display Character
;;       09H     Display String
;;       3DH     Open a File
;;       3EH     Close a File Handle
;;       3FH     Read From File/Device
;;       4CH     Terminate a Process

CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG

NERRMES DB      0DH,0AH,"Not found",0DH,
CSEG    ENDS
        END     START

A>
```

FREADプログラムを実行して、自分自身のソースファイルFREAD.ASMを読み出してみる

MS-DOSのTYPEコマンドと同様に、ファイルの内容が表示される

図2.3　完成したFREADプログラムの実行例

リスト 2.1 でも行っているように、ファイルを読み出すには、**図 2.4** の 3 つの手順が必要です。

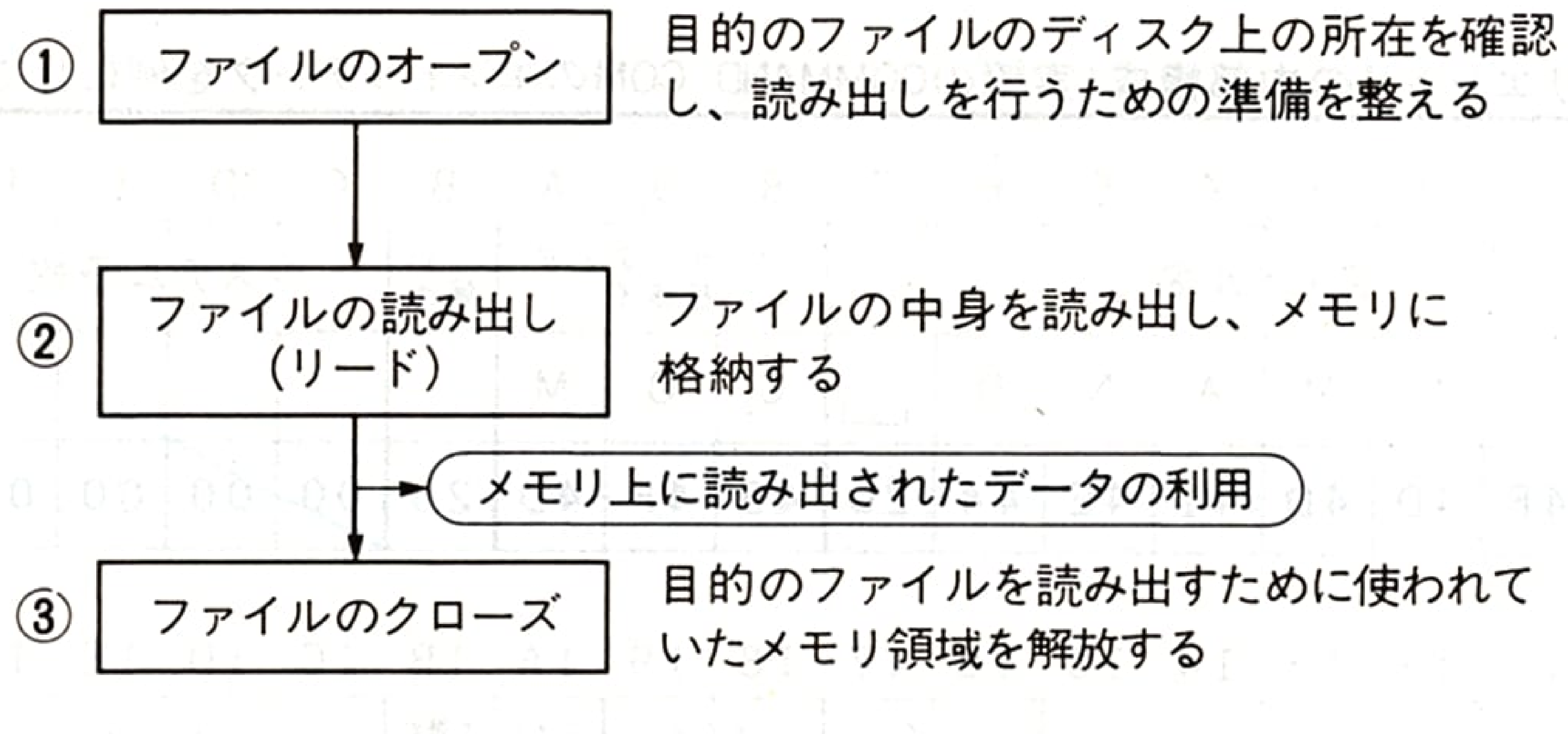

①
ファイルのオープン
目的のファイルのディスク上の所在を確認し、読み出しを行うための準備を整える
②
ファイルの読み出し（リード）
ファイルの中身を読み出し、メモリに格納する
メモリ上に読み出されたデータの利用
③
ファイルのクローズ
目的のファイルを読み出すために使われていたメモリ領域を解放する

図 2.4　ファイルを読み出す手順

FREAD プログラムも、この手順どおりにプログラムされています。この 3 つの手順に注目して、再度ソースファイルの流れを読んでみてください（なおこのプログラムは、4 章のリスト 4.3 でも取り上げて、さらに詳しい解説を行う）。

では、これらの動作によって何が行われ、それがどのような働きをするのか、その過程を追いながら、MS-DOS のファイルシステムの仕組みを見ていきましょう。ただしここでは、ファイルの読み出し操作を対象に解説し、書き込みについては触れていない部分もありますので注意してください。

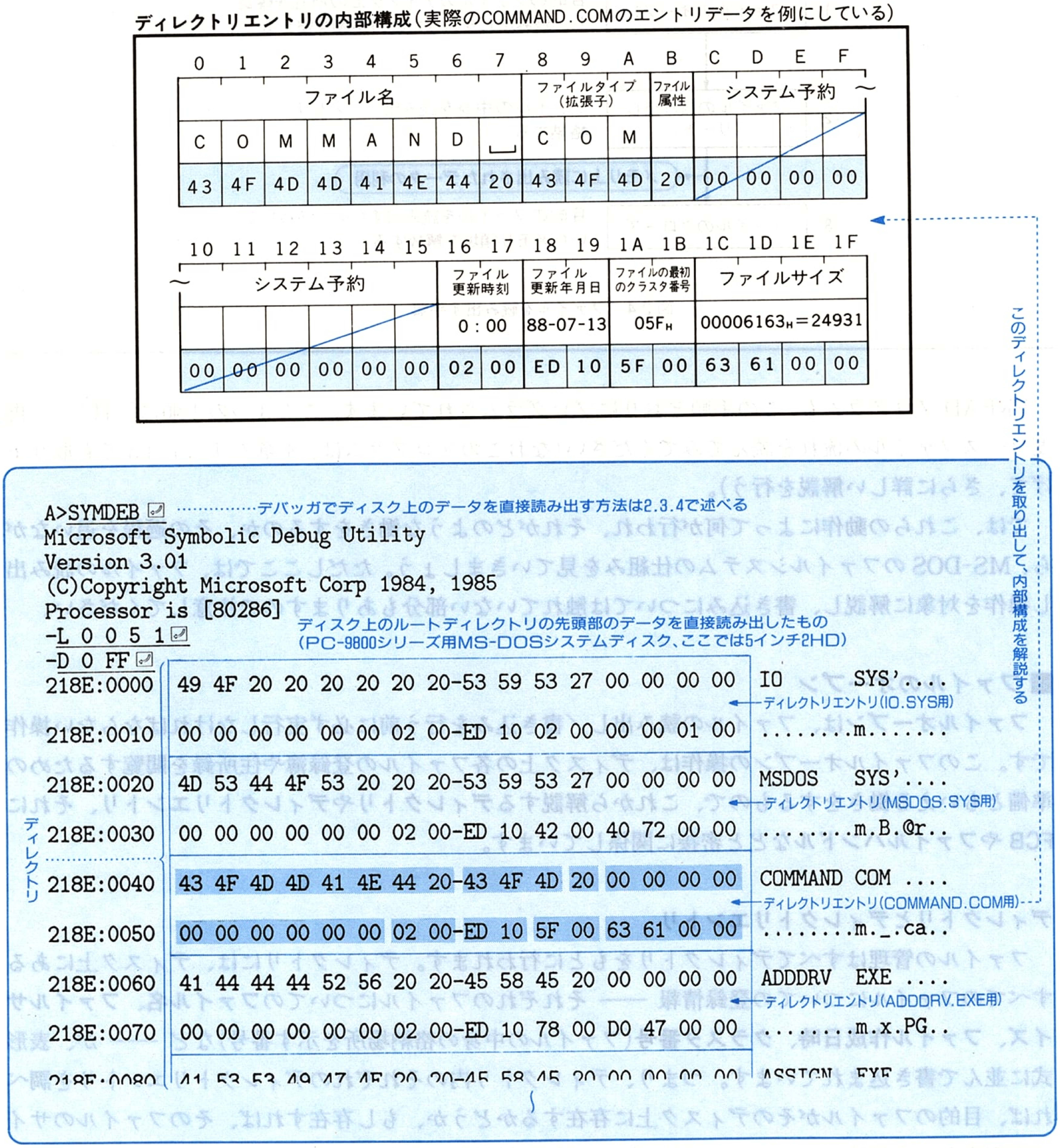
■ ファイルのオープン

ファイルオープンは、ファイルの読み出し／書き込みを行う前に必ず実行しなければならない操作です。このファイルオープンの操作は、ディスク上の各ファイルの登録簿や住所録を閲覧するための準備ともいえる働きをするもので、これから解説する**ディレクトリ**や**ディレクトリエントリ**、それに **FCB** や**ファイルハンドル**などと密接に関係しています。

ディレクトリとディレクトリエントリ

ファイルの管理はすべてディレクトリをもとに行われます。ディレクトリには、ディスク上にあるすべてのファイルについての登録情報 —— それぞれのファイルについての**ファイル名**、**ファイルサイズ**、**ファイル作成日時**、**クラスタ番号**（ファイルの中身の格納場所を示す番号）など —— が、表形式に並んで書き込まれています。つまり、ディレクトリ内のそれぞれのディレクトリエントリを調べれば、目的のファイルがそのディスク上に存在するかどうか、もし存在すれば、そのファイルのサイズや変更された日時、ファイルの本体（中身）の格納場所などを知ることができるのです。私たちが日

常最もよく使うDIRコマンドは、まさにこのディレクトリを読み出して、その中の必要な情報を一覧表の形式に整えてディスプレイに表示するコマンドなのです。なお、この登録情報を**ディレクトリエントリ**と呼びます（図2.5参照）。



ディレクトリエントリの内部構成（実際のCOMMAND.COMのエントリデータを例にしている）

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ファイル名								ファイルタイプ（拡張子）			ファイル属性	システム予約 ～			
C	O	M	M	A	N	D	␣	C	O	M					
43	4F	4D	4D	41	4E	44	20	43	4F	4D	20	00	00	00	00

10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
～ システム予約						ファイル更新時刻		ファイル更新年月日		ファイルの最初のクラスタ番号		ファイルサイズ			
						0：00		88-07-13		$05F_{H}$		$00006163_{H}=24931$			
00	00	00	00	00	00	02	00	ED	10	5F	00	63	61	00	00

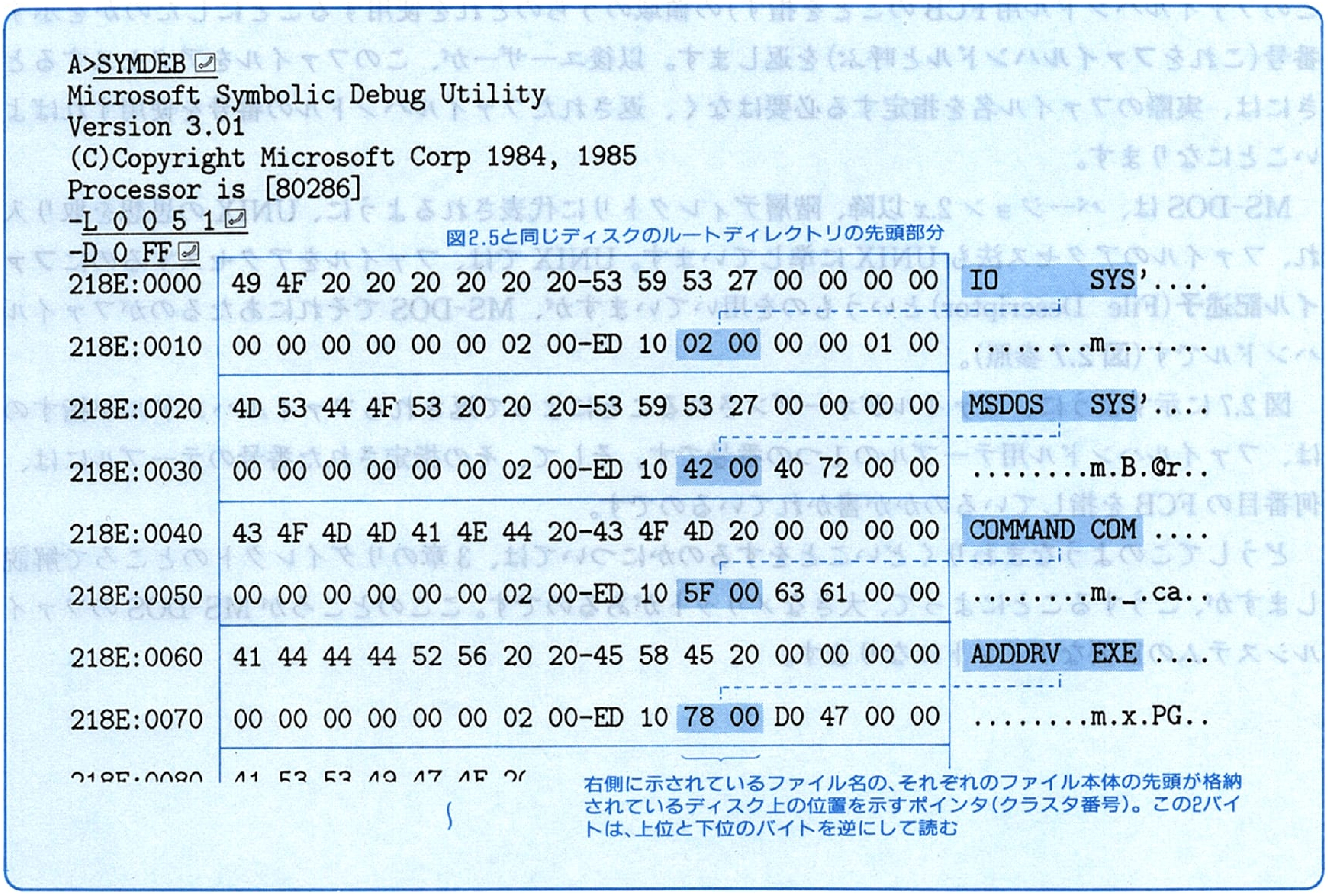
```
A>SYMDEB↵
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1↵
-D 0 FF↵
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  41 44 44 44 52 56 20 20-45 58 45 20 00 00 00 00  ADDDRV  EXE ....
218E:0070  00 00 00 00 00 00 02 00-ED 10 78 00 D0 47 00 00  ........m.x.PG..
218E:0080  41 53 53 49 47 4E 20 20-45 58 45 20 00 00 00 00  ASSIGN  EXE
```

デバッガでディスク上のデータを直接読み出す方法は2.3.4で述べる
ディスク上のルートディレクトリの先頭部のデータを直接読み出したもの（PC-9800シリーズ用MS-DOSシステムディスク、ここでは5インチ2HD）
ディレクトリエントリ(IO.SYS用)
ディレクトリエントリ(MSDOS.SYS用)
ディレクトリエントリ(COMMAND.COM用)
ディレクトリエントリ(ADDDRV.EXE用)
ディレクトリ
このディレクトリエントリを取り出して、内部構成を解説する

図2.5　**ディレクトリとディレクトリエントリの内部構成**

さて、サンプルプログラムFREADでは、まずファイルをオープンするためのファンクションリクエスト(ファンクション3DH)によって、ディレクトリの中から読み出そうとする目的のファイルが捜し出され、もし見つからなければ、「ファイルが存在しない」というエラー情報が返されます。目的のファイルが見つかった場合は、そのファイルをアクセスする(読み出したり書き込んだりすること)ための準備が行われます。ファンクションリクエストについては4章で詳しく解説しますが、ここではとりあえず、ファイルをオープンする、中身を読み出す、ディスプレイに文字を表示する、キーボードから文字を入力するなどの、MS-DOSの数十種類に及ぶ基本機能を、ユーザープログラムで利用するための「機能の呼び出し」と考えておいてください。

ファイルをアクセスするためには、まず目的のファイルの本体が、ディスク上のどこに格納されているかを知らなくてはなりません。そのために、ディレクトリ内の各ディレクトリエントリには、それぞれのファイルの本体がディスク上のどこに格納されているかを示すポインタ(後述の**クラスタ番号**)が用意されており、このポインタを参照することにより、ファイルの本体にたどり着くことができるようになっています(**図2.6**参照)。ただし、ファイルサイズが0である場合には、存在するのは名前だけで、ファイルの本体は存在していません。



```
A>SYMDEB↵
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1↵
-D 0 FF↵
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  41 44 44 44 52 56 20 20-45 58 45 20 00 00 00 00  ADDDRV  EXE ....
218E:0070  00 00 00 00 00 00 02 00-ED 10 78 00 D0 47 00 00  ........m.x.PG..
218E:0080  41 53 53 49 47 4E 2
```

図2.5と同じディスクのルートディレクトリの先頭部分
右側に示されているファイル名の、それぞれのファイル本体の先頭が格納されているディスク上の位置を示すポインタ(クラスタ番号)。この2バイトは、上位と下位のバイトを逆にして読む

図2.6　ディレクトリエントリにファイルの格納場所が示されている

FCB(ファイルコントロール・ブロック)

さて、目的のファイルがディスク上に存在していることが判明した場合、次はそのファイルの読み出しあるいは書き込み操作のための処理を行わなければなりません。たとえば、ここでのサンプルプログラムFREADのように、ファイルを先頭から順に読み出していく場合には、MS-DOSは、ファイルを現在どこまで読み出したか、またファイルの最終はどこなのかなどを管理する必要があり、そのための「テーブル」(表)が作成されます。**FCB**(File Control Block)と呼ばれるこのテーブルは、目的のファイルのディレクトリエントリの内容が、メモリ上のFCBの領域に自動的にコピーされることにより作成されます。

ファイルをアクセスする手順の最初の操作であるファイルのオープンとは、ディレクトリの中から目的のファイルを捜し出し、もし存在すれば、そのファイルをアクセスするための管理に必要なデータをメモリ上に用意することだったのです。

ファイルハンドル

ユーザープログラムにおいて、ファイル名を指定することによって目的のファイルがオープンされると、MS-DOSシステムは、いくつかあるファイルハンドル用FCB(以降、単にFCBと記すものは、このファイルハンドル用FCBのことを指す)の領域のうちのどれを使用することにしたのかを示す番号(これを**ファイルハンドル**と呼ぶ)を返します。以後ユーザーが、このファイルをアクセスするときには、実際のファイル名を指定する必要はなく、返されたファイルハンドルの番号を使用すればよいことになります。

MS-DOSは、バージョン2.*x*以降、階層ディレクトリに代表されるように、UNIXの思想を取り入れ、ファイルのアクセス法もUNIXに準じています。UNIXでは、ファイルをアクセスするのに**ファイル記述子**(File Descriptor)というものを用いていますが、MS-DOSでそれにあたるのがファイルハンドルです(**図2.7**参照)。

図2.7に示すように、ファイルがオープンされることによって返されるファイルハンドルが指すのは、ファイルハンドル用テーブルの1つの番号です。そして、その指定された番号のテーブルには、何番目のFCBを指しているのかが書かれているのです。

どうしてこのようなまわりくどいことをするのかについては、3章のリダイレクトのところで解説しますが、こうすることによって、大きなメリットがあるのです。ここのところがMS-DOSのファイルシステムの重要なポイントになります。

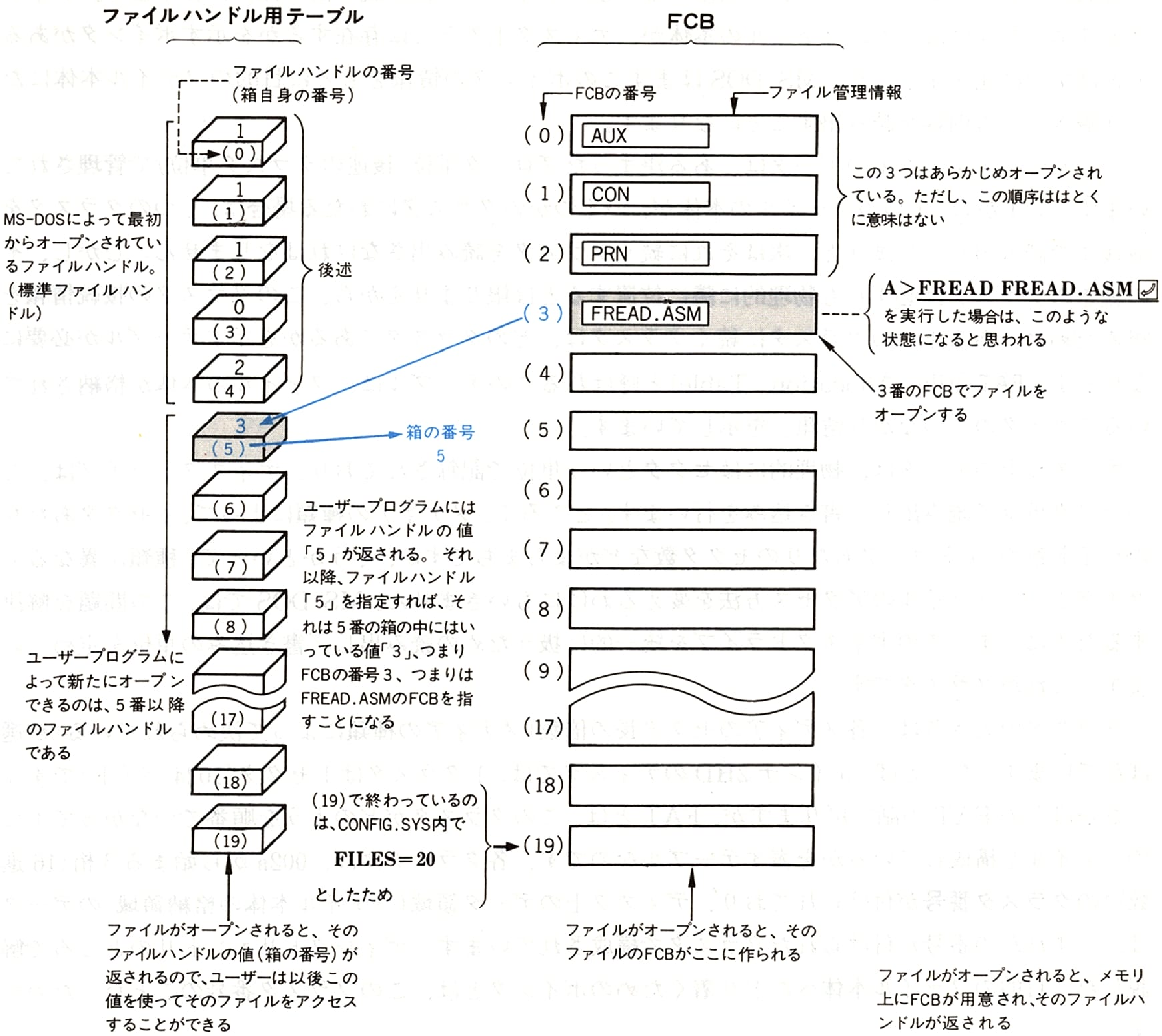

ファイルハンドル用テーブル
FCB
ファイルハンドルの番号
（箱自身の番号）
FCBの番号
ファイル管理情報
1 (0)
1 (1)
1 (2)
0 (3)
2 (4)
3 (5)
(6)
(7)
(8)
(17)
(18)
(19)
(0) AUX
(1) CON
(2) PRN
(3) FREAD.ASM
(4)
(5)
(6)
(7)
(8)
(9)
(17)
(18)
(19)
MS-DOSによって最初からオープンされているファイルハンドル。（標準ファイルハンドル）
後述
この3つはあらかじめオープンされている。ただし、この順序ははとくに意味はない
A>FREAD FREAD.ASM⏎
を実行した場合は、このような状態になると思われる
3番のFCBでファイルをオープンする
箱の番号
5
ユーザープログラムにはファイルハンドルの値「5」が返される。それ以降、ファイルハンドル「5」を指定すれば、それは5番の箱の中にはいっている値「3」、つまりFCBの番号3、つまりはFREAD.ASMのFCBを指すことになる
ユーザープログラムによって新たにオープンできるのは、5番以降のファイルハンドルである
(19)で終わっているのは、CONFIG.SYS内で
FILES=20
としたため
ファイルがオープンされると、そのファイルハンドルの値（箱の番号）が返されるので、ユーザーは以後この値を使ってそのファイルをアクセスすることができる
ファイルがオープンされると、そのファイルのFCBがここに作られる
ファイルがオープンされると、メモリ上にFCBが用意され、そのファイルハンドルが返される

図 2.7　ファイルハンドルと FCB

■ ファイルの読み出し（リード）

目的のファイルの本体を読み出すのがファイルの読み出しです。さきほどお話ししたファイルオープンの操作によって目的のファイルが存在することが確認され、その登録簿や住所録に相当する FCB が取り出されて、メモリ上に用意されます。問題は、MS-DOS がこのあとどのようにしてディスク上の目的のファイル本体が格納されている場所を捜し当て、そこから中身を読み出すか、ということです。

FAT（ファイルアロケーション・テーブル）とクラスタ

目的のファイルがオープンされ、次はいよいよファイルの本体を読み出す操作に移ります。ディレクトリエントリには、そのファイルの本体が、ディスク上のどこに存在するかを示すポインタがあることはすでに述べましたが、MS-DOSはまずこのポインタの情報をもとに目的のファイル本体にたどり着き、その内容を読み出すことになります。

ところで、ディスク上のデータは、ある決まったブロック単位（後述のクラスタ単位）で管理されています。ですから、1つのファイルの本体が、いくつかのクラスタにわたる場合は、1つのクラスタを最後まで読み出してしまうと、次はそれに続くクラスタを読み出さなければなりません。しかし、その続きのクラスタは必ずしも物理的に隣に位置するとは限りませんから、このクラスタの接続情報を得るために、ある1つのクラスタに続くクラスタは、どのクラスタであるかを示すテーブルが必要になります。**FAT**（File Allocation Table）と呼ばれるこのテーブルは、ファイルの本体が格納されているクラスタの「つながり情報」を示しています。

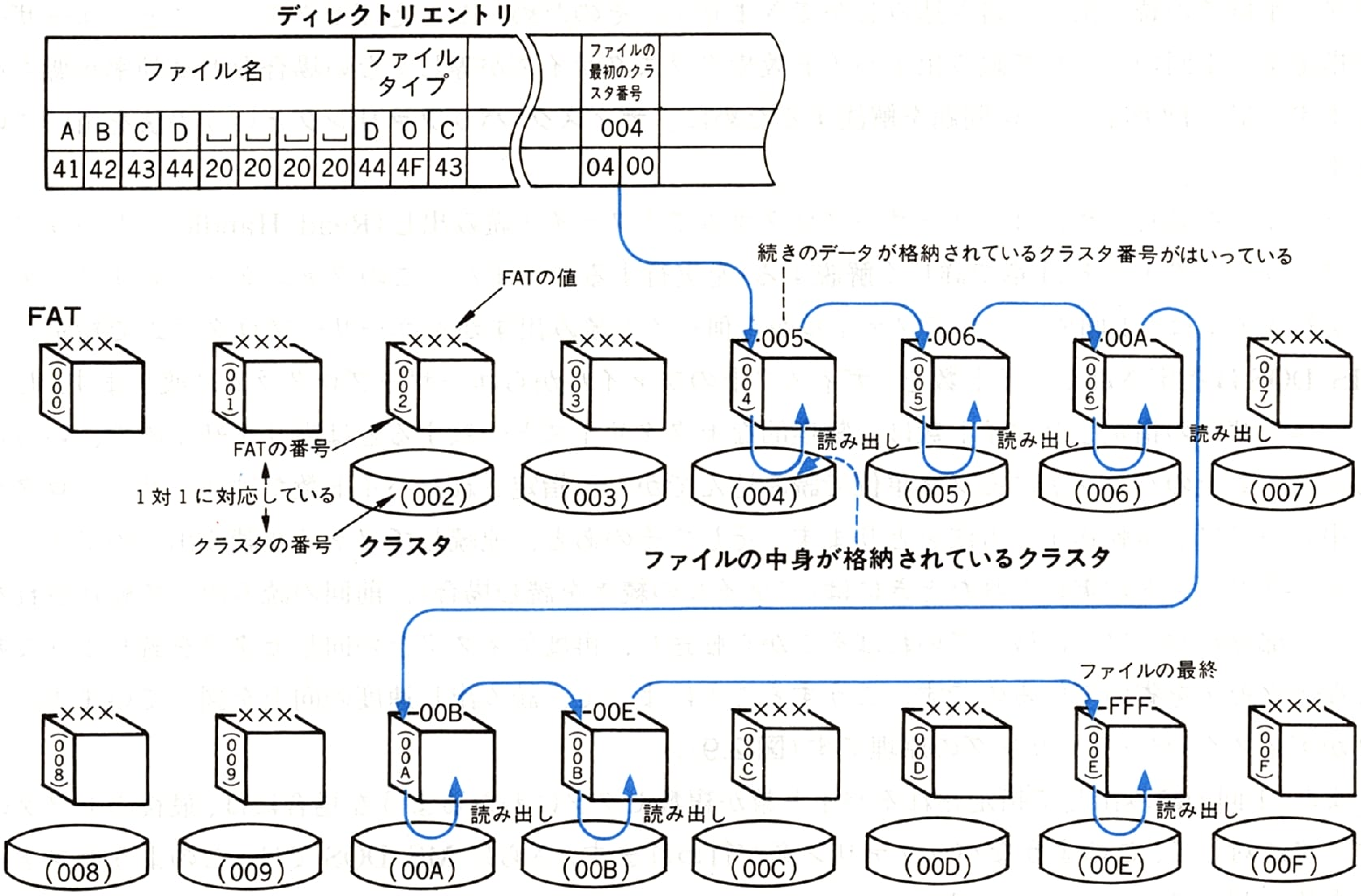
ディスク上のデータは、物理的には**セクタ**という単位で記録されており、ディスクドライブは、このセクタ単位で読み出し／書き込みを行います。ところで、メディアの種類によって、1セクタあたりのバイト数や、1トラックあたりのセクタ数などがまちまちですが、そうかといって、種類が異なるドライブごとにファイルのアクセス方法を変えるわけにもいきません。MS-DOSでは、この問題を解決するために、すべてのディスクドライブを統一的に扱うための読み出し／書き込みの単位を定めています。これがクラスタです。

クラスタの大きさは、各メディアのセクタ長の倍数（メディアの種類によって決められている）が選ばれています。たとえば、5インチ2HDのディスクでは、1クラスタは1セクタ（1024バイト）です。

さきほどのFATの話に戻りますが、FATとは、このクラスタがどのような順番でつながって1つのファイルを構成しているかを表すテーブルなのです。各クラスタには、002Hから始まる3桁（16進数）*の**クラスタ番号**が付けられており、ディスク上のデータ領域（ファイル本体の格納領域）のデータは、いずれかの番号が付けられたクラスタで構成されています。ディレクトリエントリのところで解説した、目的のファイル本体へたどり着くためのポインタとは、このクラスタ番号のことだったのです。

FATは、ディスク上に存在するクラスタの数と同じ数のテーブルであり、たとえば5番目のクラスタに続くクラスタの番号は、それに対応しているFAT（やはり5番目）に書かれています（**図2.8**参照）。

＊ 2.3.9で解説するように、ハードディスクなどで16ビットFATを利用している場合は、4桁のクラスタ番号が付けられる。


ディレクトリエントリ
ファイル名
ファイルタイプ
ファイルの最初のクラスタ番号
A B C D ␣ ␣ ␣ ␣ D O C
41 42 43 44 20 20 20 20 44 4F 43
004
04 00
続きのデータが格納されているクラスタ番号がはいっている
FATの値
FAT
×××
005
006
00A
00B
00E
FFF
ファイルの最終
(000) (001) (002) (003) (004) (005) (006) (007)
(008) (009) (00A) (00B) (00C) (00D) (00E) (00F)
FATの番号
1対1に対応している
クラスタの番号
クラスタ
読み出し
ファイルの中身が格納されているクラスタ

FATの情報により、ファイルABCD.DOCの中身は、クラスタ番号、$004_H \rightarrow 005_H \rightarrow 006_H \rightarrow 00A_H \rightarrow 00B_H \rightarrow 00E_H$の順にディスクに格納されていることがわかる。このクラスタ番号の順を記したものが、ファイルアロケーションテーブル(FAT)である

図 2.8　クラスタが接続される原理とディレクトリ、FAT、クラスタの関係

図 2.8 にあるように、ファイルの最後のクラスタに対応する FAT には、ファイルの最後を示すための特別な番号(通常は $\mathrm{FFF_H}$)が書き込まれています。

さて、FAT とクラスタの話は、ちょっとむずかしくなりましたが、目的のファイルを読み出すという操作が、そのファイルの FAT をたどることによって行われていることが、だいたい理解できたでしょうか。

■ディスク・バッファリング

すでに解説したように、ディスクから実際にデータを読み出す場合、ディスクドライブは原理的にセクタ単位での読み出し／書き込みしかできません。そのため、たとえばセクタサイズと、ユーザーが指定する1回のコールで読み出すバイト数やクラスタサイズが等しくない場合などは効率が悪くなります。MS-DOSは、この問題を解決するために、**ディスク・バッファリング**という手法を用いています。

ファイルを読むためには、ユーザープログラムで「ファイル読み出し(Read Handle)」というファンクションリクエスト(4章で詳しく解説する)を実行するのですが、このファンクションリクエストを実行するには、1回のコールでファイルから何バイト読み出すかをユーザープログラムで指定し、MS-DOSは指定されたバイト数を、ディスク上のファイルからユーザープログラムに渡します。しかし、ユーザーの指定したバイト数は、物理的なセクタサイズと一致するとは限りませんので、いったんシステム上のバッファにセクタ単位で読み込んでから、指定されたバイト数だけユーザープログラム中のバッファへ転送する方法をとります。そしてそのあと、連続してファイル読み出しのファンクションリクエストが実行されたときには(ファイルの続きを読む場合)、前回の読み出しで転送されなかった部分がバッファに残っていればそこから転送し、再度ディスク上の同じセクタを読むようなむだなアクセスを省いているのです。こうすることによって、読み出し速度の向上を図っています。これがディスク・バッファリングの原理です(**図2.9**)。

また、1回の読み出しで指定されるバイト数が複数セクタにわたるような場合には、最後のセクタのデータに対して、このようなバッファリングが行われます。さらにMS-DOSでは、このようなファイル本体に対するバッファリングのほかに、ディレクトリやFATの読み出し／書き込みに対してもバッファリングを行っていますが、これについては後述します。

さて、あるファイルを読み進んでいくとしましょう。最初のセクタを最後まで読んでしまうと次のセクタへと進みます。さらに読み進んで、最初のクラスタの最後のセクタを読み終わると、結局最初のクラスタ全部が読み出されたことになり、次はこれに続くクラスタを読み出さなければなりません。その際、現在のクラスタに対応するFATを参照することにより、次はどのクラスタを読めばよいかがわかります。

MS-DOSは、バッファリングされたデータのどこまでをユーザーが取り出したか、いつ次のセクタを読み出すか、いつ次のクラスタへ進まなければならないか、どこがファイルの最後かなどを管理するために、ファイルのサイズやファイルの読み出し状況などの情報を常に保持しておかなければなりません。そのための作業領域がFCBなのです。つまりファイルの読み出しとは、ディスクからデータを読み出すとともに、FCBの情報を次回の読み出し動作に備えて更新することでもあります。

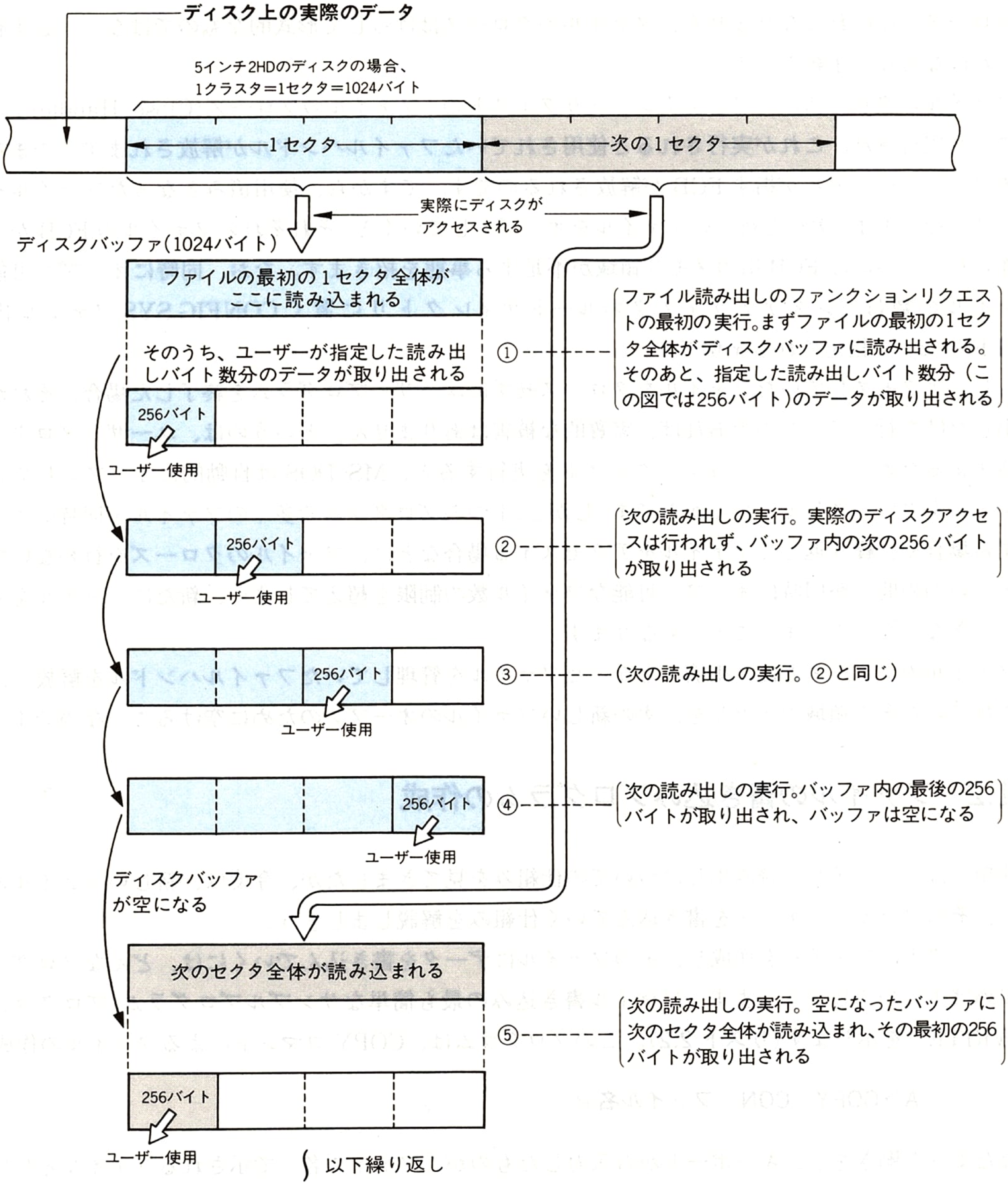

ディスク上の実際のデータ
5インチ2HDのディスクの場合、
1クラスタ＝1セクタ＝1024バイト
1セクタ
次の1セクタ
実際にディスクが
アクセスされる
ディスクバッファ(1024バイト)
ファイルの最初の1セクタ全体が
ここに読み込まれる
そのうち、ユーザーが指定した読み出
しバイト数分のデータが取り出される
256バイト
ユーザー使用
①
ファイル読み出しのファンクションリクエストの最初の実行。まずファイルの最初の1セクタ全体がディスクバッファに読み出される。そのあと、指定した読み出しバイト数分（この図では256バイト）のデータが取り出される
256バイト
ユーザー使用
②
次の読み出しの実行。実際のディスクアクセスは行われず、バッファ内の次の256バイトが取り出される
256バイト
ユーザー使用
③
（次の読み出しの実行。②と同じ）
256バイト
ユーザー使用
④
次の読み出しの実行。バッファ内の最後の256バイトが取り出され、バッファは空になる
ディスクバッファ
が空になる
次のセクタ全体が読み込まれる
⑤
次の読み出しの実行。空になったバッファに次のセクタ全体が読み込まれ、その最初の256バイトが取り出される
256バイト
ユーザー使用
以下繰り返し

図 2.9 ファイル読み出し時のディスク・バッファリングの原理

■ ファイルのクローズ

目的のファイルを読み終わり、そのファイルに関する仕事がすべて終了したならば、そのファイルをクローズしなければなりません。ファイルのクローズはけっして形式的なものではなく、必ず行わなければならない手続きです。

ファイルのクローズは、ファンクションリクエストの「ファイルのクローズ(Close Handle)」の機能により実行され、これが実行されると使用されていたファイルハンドルが解放されます。つまり、そのファイルハンドルが指すFCBが解放されるのです。ですから、使用済みとなったファイルをクローズしないまま、次々と新しいファイルをオープンしていくと、それぞれのファイルのFCBが次々と作られ、ついにはFCB用のメモリ領域が不足する事態を招きます。なお、同時にオープン可能なファイルの数は、システムディスクのルートディレクトリに置くCONFIG.SYSファイル内の「FILES＝x」で指定されるx個です。

もし、アクセスしていたファイルをクローズせずにユーザープログラムを終了した場合、それが読み出しだけを行っていたのであれば、実質的な被害はありません。というのは、ユーザープログラムを終了するためのファンクションリクエストを実行すると、MS-DOSは自動的にオープンしているファイルをクローズしてくれるからです。しかし、1つのプログラムで多くのファイルを同時にアクセスする場合や、繰り返してファイルをアクセスする場合などに、ファイルのクローズを行わないでいると、いつの間にか同時にオープン可能なファイル数の制限を超えてしまい、新たにファイルをオープンできなくなってしまうことにもなります。

ファイルのクローズとは、用済みとなったファイルを管理していたファイルハンドルを解放し、占有されていたその領域のメモリを、次の新しいファイルのオープンのために空けることなのです。

2.1.2　ファイルの書き込みプログラムの作成

前項では、ファイルの読み出しについての仕組みを見てきましたが、今度は、新しいファイルの作成と、そのファイルにデータを書き込んでいく仕組みを解説しましょう。

ディスク上にファイルを作成し、そのファイルにデータを書き込んでいくには、どんなプログラムを組めばよいのでしょう。まず、ファイル書き込みの最も簡単なサンプルプログラム(プログラム名「FWRITE」)を示します(**リスト2.2**)。このプログラムは、COPYコマンドによるファイルの作成

```
A>COPY  CON  ファイル名↵
```

と似たような働きをし、キーボードから入力したものが「ファイル名」で示されるファイルとして作成されます。

ファイルの書き込みに関する動作は、読み出しの場合と逆の動作をするものが多く、前項と同じような部分も出てきますが、ここではそれらがファイルの書き込みのために動作することになります。

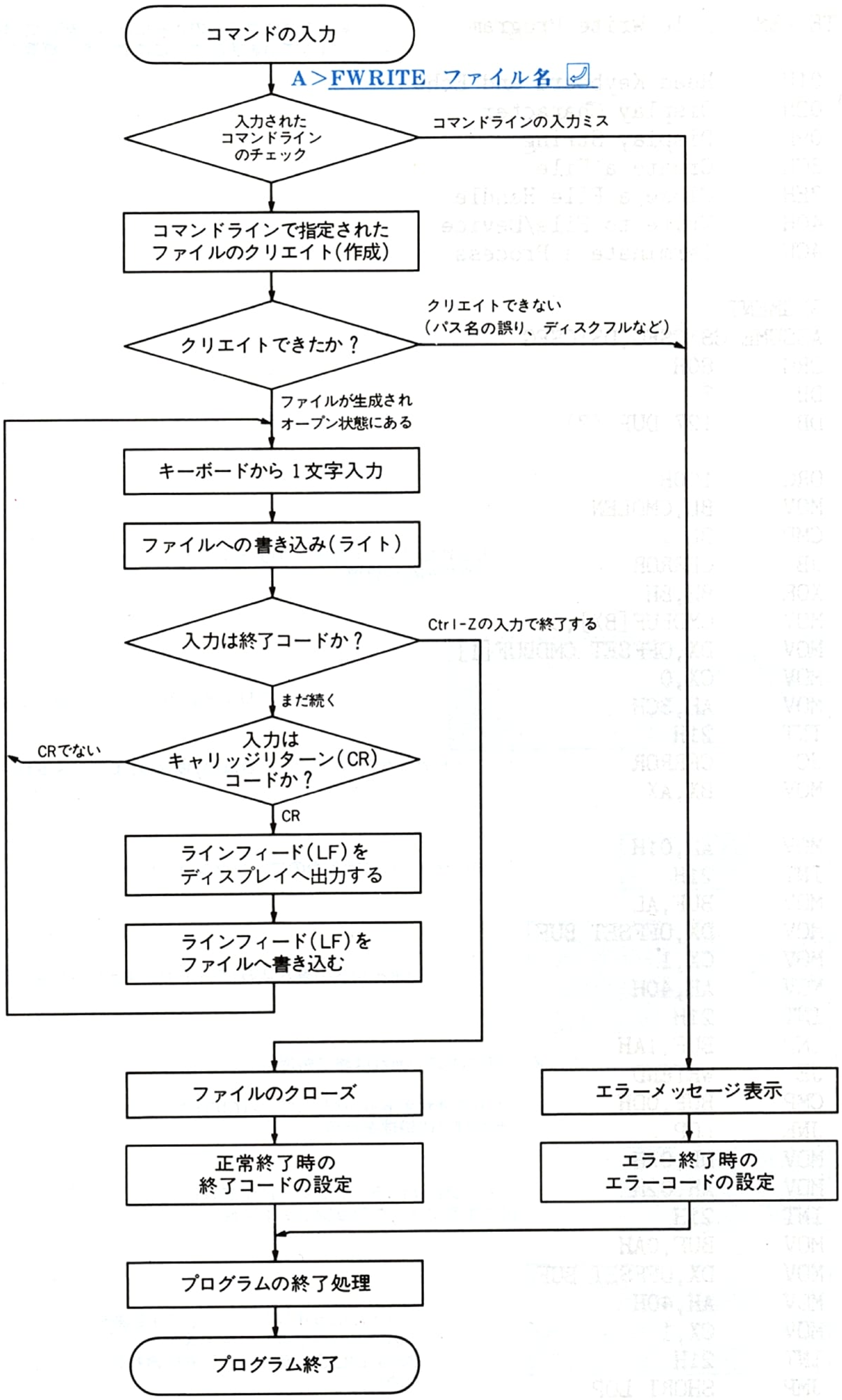

コマンドの入力
A>FWRITE ファイル名 ↵
入力された
コマンドライン
のチェック
コマンドラインの入力ミス
コマンドラインで指定された
ファイルのクリエイト(作成)
クリエイトできない
(パス名の誤り、ディスクフルなど)
クリエイトできたか？
ファイルが生成され
オープン状態にある
キーボードから１文字入力
ファイルへの書き込み(ライト)
Ctrl-Zの入力で終了する
入力は終了コードか？
まだ続く
CRでない
入力は
キャリッジリターン(CR)
コードか？
CR
ラインフィード(LF)を
ディスプレイへ出力する
ラインフィード(LF)を
ファイルへ書き込む
ファイルのクローズ
エラーメッセージ表示
正常終了時の
終了コードの設定
エラー終了時の
エラーコードの設定
プログラムの終了処理
プログラム終了

図 2.10　FWRITE プログラムの構成

＊このプログラムのさらに詳しい解説は、4章のリスト4.4で行っていますので、ここではその概要を示します

```
;; FWRITE.ASM     File Write Program
;;
;;        01H     Read Keyboard and Echo
;;        02H     Display Character
;;        09H     Display String
;;        3CH     Create a File
;;        3EH     Close a File Handle
;;        40H     Write to File/Device
;;        4CH     Terminate a Process

CSEG      SEGMENT
          ASSUME CS:CSEG,DS:CSEG
          ORG     80H
CMDLEN    DB      ?
CMDBUF    DB      127 DUP (?)

          ORG     100H
START:    MOV     BL,CMDLEN
          CMP     BL,2
          JB      CERROR
          XOR     BH,BH
          MOV     CMDBUF[BX],0
          MOV     DX,OFFSET CMDBUF[1]
          MOV     CX,0
          MOV     AH,3CH
          INT     21H
          JC      CERROR
          MOV     BX,AX

LOP:      MOV     AH,01H
          INT     21H
          MOV     BUF,AL
          MOV     DX,OFFSET BUF
          MOV     CX,1
          MOV     AH,40H
          INT     21H
          CMP     BUF,1AH
          JE      WRTEND
          CMP     BUF,0DH
          JNE     LOP
          MOV     DL,0AH
          MOV     AH,02H
          INT     21H
          MOV     BUF,0AH
          MOV     DX,OFFSET BUF
          MOV     AH,40H
          MOV     CX,1
          INT     21H
          JMP     SHORT LOP
```

入力されたコマンドラインのチェックと、その文字列の前処理

……ファイルクリエイトのファンクションリクエスト

……クリエイトできない(オープンできない)場合は、エラーメッセージの表示へ

……キーボードから1文字入力のファンクションリクエスト

……ファイルへの書き込み(ライト)のファンクションリクエスト

入力文字がCtrl-Zであれば終了処理へ

……入力文字がキャリッジリターン(CR)であればこの処理を行う

……ディスプレイにラインフィード(LF)を出力するファンクションリクエスト

……ファイルにラインフィード(LF)を書き込むファンクションリクエスト(MS-DOSのテキスト形式に合わせるため)

繰り返し

— リスト 2.2 — (次ページに続く)

```
WRTEND: MOV     AH,3EH      ……………ファイルクローズのファンクションリクエスト
        INT     21H
        XOR     AL,AL     ……………正常終了のコードを設定する
        JMP     SHORT RETURN

CERROR: MOV     DX,OFFSET CERRMES
        MOV     AH,09H            ……………エラーメッセージを出力するファンクションリクエスト
        INT     21H
        MOV     AL,1    ……………エラーコードを設定する
RETURN: MOV     AH,4CH
        INT     21H         ……………プログラムの終了処理

BUF     DB      0
CERRMES DB      0DH,0AH,"Can't create",0DH,0AH,'$' ……………エラーメッセージ文字列

CSEG    ENDS
        END     START
```

リスト 2.2　ファイルの書き込みプログラム FWRITE のソースファイル

アセンブルおよびリンク作業は、前項のファイルの読み出しプログラムの場合と同じですので、その実行例は省略します。FWRITE プログラムの使い方は、**図 2.11** を参照してください。

```
A>FWRITE TEST↵ ……………FWRITEプログラムを実行して、新しいファイルTESTを作成する

This is a test for "OUYOU MS-DOS".↵ ┐
↵                                   │ファイルの内容を入力していく
Can you read me ?^Z                 ┘
                 ↑————最後にはCtrl-Zを入力する(PC-9800シリーズでは画面はクリアされる)
A>TYPE TEST↵ ……………作成されたファイルTESTの内容を表示してみる
This is a test for "OUYOU MS-DOS".  ┐
                                    │入力した内容のファイルが作成されていた
Can you read me ?                   ┘

A>
```

図 2.11　FWRITE プログラムの実行例

リスト 2.2 で行っているように、新しいファイルを作成するには図 2.12 の 3 つの手順が必要です。

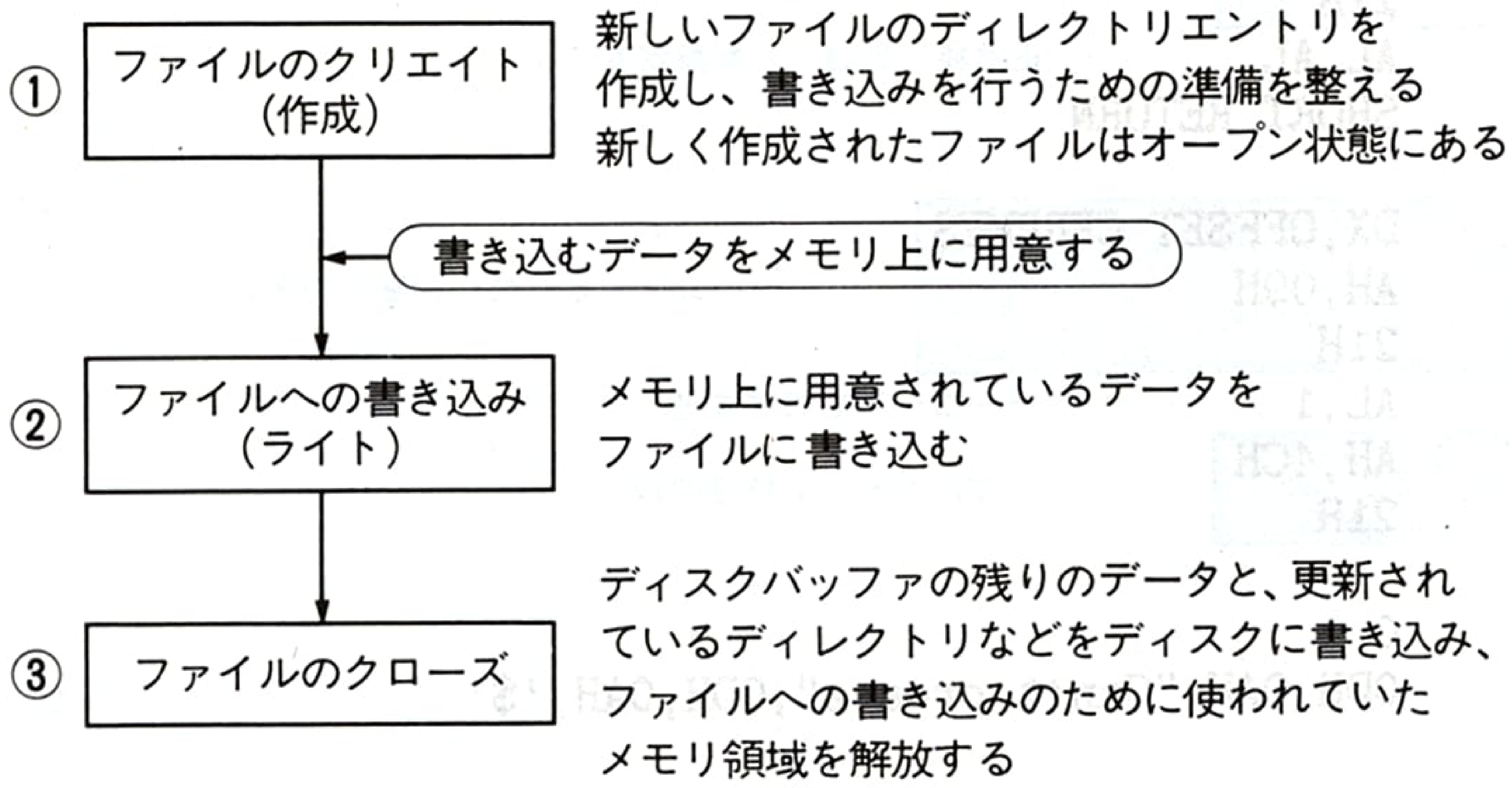

①
ファイルのクリエイト
（作成）
新しいファイルのディレクトリエントリを
作成し、書き込みを行うための準備を整える
新しく作成されたファイルはオープン状態にある
書き込むデータをメモリ上に用意する
②
ファイルへの書き込み
（ライト）
メモリ上に用意されているデータを
ファイルに書き込む
③
ファイルのクローズ
ディスクバッファの残りのデータと、更新され
ているディレクトリなどをディスクに書き込み、
ファイルへの書き込みのために使われていた
メモリ領域を解放する

図2.12　新しいファイルを作成する手順

■ファイルのクリエイト（新規ファイルの作成）

ここでまず、既存のファイルを書き込みモードでオープンすることを考えましょう。前項では触れませんでしたが、この**モード**とはファイルをオープンする目的を示す名称のことで「読み出し」「書き込み」「読み出し／書き込み」などがあり、ファイルオープンの際には、その他のパラメータとともに指定しなければなりません（リスト2.2参照）。

書き込みモードでのオープンは、基本的には読み出しモードでのオープンと同様の処理が行われます。与えられたファイル名によってディレクトリを検索し、そのファイルが存在していれば新しいFCBを確保して、そのファイルハンドルの番号を返すのです。ということは、書き込みモードでファイルをオープンする場合は、ディスク上にすでに存在しているファイルでなければオープンすることはできない、つまり、既存ファイルの更新しかできないことになります。

では、新しいファイルを作成する場合はどのような操作をすればよいのでしょう。新しいファイルを作成する場合には、「ファイルのクリエイト（Create Handle）」のファンクションリクエストを実行します。この機能は、指定したファイルがディスク上に存在している場合でも有効で、その場合は書き込みモードでのオープンとあまり変わりませんが、もとのファイルのサイズは強制的に0になります。つまり、ファイルのクリエイトで指定したファイル名と同じ名前のファイルがすでに存在していた場合は、その内容がすべて削除されてしまうのです。これは、使い方を誤ると恐ろしいことになりますので注意してください。もし指定したファイルが存在しない場合には、新しいファイルを作成する処理が行われます（図2.13）。

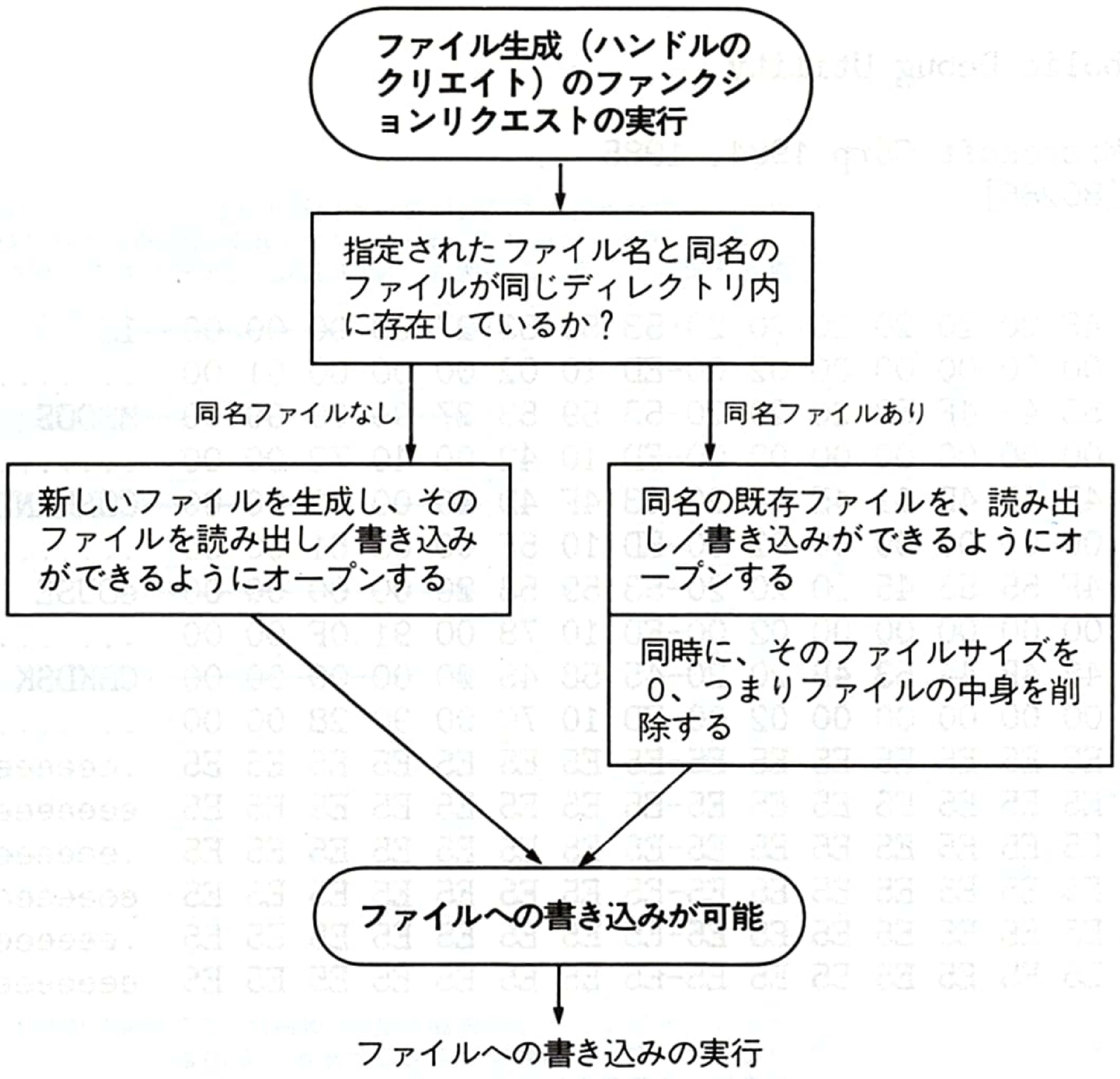

ファイル生成（ハンドルのクリエイト）のファンクションリクエストの実行
指定されたファイル名と同名のファイルが同じディレクトリ内に存在しているか？
同名ファイルなし
同名ファイルあり
新しいファイルを生成し、そのファイルを読み出し／書き込みができるようにオープンする
同名の既存ファイルを、読み出し／書き込みができるようにオープンする
同時に、そのファイルサイズを0、つまりファイルの中身を削除する
ファイルへの書き込みが可能
ファイルへの書き込みの実行

図 2.13　ファイルのクリエイトの動作原理

新しいファイルを作成するために、MS-DOS はまずディレクトリ内にそのファイルを登録するためのディレクトリエントリ用の空き領域を確保しなければなりません。現在使われていないディレクトリエントリや、消去されたファイルが使っていたディレクトリエントリの先頭には、使用されていないことを示すマークが書き込まれていますので、MS-DOS はその空きマークを捜して、そこを新しいファイルのディレクトリエントリに割り当てます。具体的には、新しく作成するファイル名、ファイルサイズ(最初は 0)などをそのディレクトリエントリに書き込み、それと同時に FCB が作成されて、ユーザープログラムにはその FCB のファイルハンドルの番号が返されるのです。つまり、新しいファイルが登録(作成)され、そのファイルがオープンされた状態になります(図 2.14 参照)。

```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  E5 4F 55 53 45 20 20 20-53 59 53 20 00 00 00 00  eOUSE   SYS ....
218E:0070  00 00 00 00 00 00 02 00-ED 10 78 00 91 0F 00 00  ........m.x.....
218E:0080  43 48 4B 44 53 4B 20 20-45 58 45 20 00 00 00 00  CHKDSK  EXE ....
218E:0090  00 00 00 00 00 00 02 00-ED 10 7C 00 90 28 00 00  ........m.|..(..
218E:00A0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:00B0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
218E:00C0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:00D0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
218E:00E0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:00F0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
-
```

ファイルが消去されて「空」になっているディレクトリエントリの先頭（つまり、ディレクトリエントリのファイル名の先頭文字）にはE5$_H$が書き込まれている。この例は、MOUSE.SYSが消去されたもの

ディレクトリエントリの先頭が00$_H$の場合は、ここが現在使われているディレクトリの最後であることを示している
つまり、このあとにはディレクトリエントリは存在していない

新しく作成されるファイルは、消去されて空になっている――先頭がE5$_H$である――ディレクトリエントリに作られる。もしそのようなエントリが存在しない場合は、未使用のエントリ（先頭が00$_H$）が使われる

```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  4E 45 57 46 49 4C 45 20-20 20 20 20 00 00 00 00  NEWFILE     ....
218E:0070  00 00 00 00 00 00 DC 12-16 13 87 00 0A 00 00 00  ......¥.........
218E:0080  43 48 4B 44 53 4B 20 20-45 58 45 20 00 00 00 00  CHKDSK  EXE ....
218E:0090  00 00 00 00 00 00 02 00-ED 10 7C 00 90 28 00 00  ........m.|..(..
218E:00A0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:00B0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
218E:00C0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
```

上に示した状態のディスク上に新しいファイルNEWFILEを作成した。消去されたファイルMOUSE.SYSが使っていた空きエントリに、今回作成されたNEWFILEが登録されている

図2.14　ファイル・クリエイト時の内部の動き

属性(アトリビュート)

ディレクトリエントリが保持するいくつかの情報の中に**属性**という項目があります。これは１章で作成したCHMODプログラムでおなじみの、リードオンリーや隠しファイルなどを表す「属性」にほかなりません。CHMODプログラムは、この属性を設定、変更するプログラムだったのです。このCHMODプログラムが実行されると、指定されたファイルのディレクトリエントリを捜すためにディスクのディレクトリが検索され、見つかるとディレクトリエントリ内の属性の項目を指定の属性に書き換える作業を行います。

さて、書き込みモードでのオープンやファイルのクリエイトを実行すると、まずこの属性がチェックされます。もし、目的のファイルにリードオンリーの属性が設定されていれば、書き込みは禁止なのでオープンは拒否されます。

■ ファイルへの書き込み(ライト)

バッファリング

目的のファイルへ書き込みを行う場合、読み出しのところで述べた動作と逆の働きをするバッファリングが行われます。書き込みのシステムコールが実行されると、ユーザープログラムで指定されたバイト数分のメモリ上のデータがファイルに書き込まれますが、実際にはそのデータを直接ディスクへ書き込むわけではありません。なぜなら、ユーザーが指定した１回のシステムコールで書き込むデータのバイト数は、ディスクのセクタサイズと一致するとは限らないからです。ディスクドライブは、物理的にはセクタ単位で読み出し／書き込みを行うため、１つのセクタのある部分だけを直接書き込んだり、書き換えたりすることはできません。このために、読み出しの場合と逆の過程をたどるディスク・バッファリングが行われます。

では、どのように書き込みが行われるのでしょうか。書き込みのシステムコールが実行されると、ユーザーの指定したメモリ上のアドレスから指定したバイト数分のデータが、MS-DOSシステムのバッファ(ディスク・バッファリングのためのバッファ)に転送されます。指定したバイト数が少なければ、最初の書き込みは、ただこれだけで終了し(つまり、ディスク上には書き込まない)、次の書き込みが実行されると、新しく書き込むデータはバッファに書き込まれている先のデータの直後に続けて転送されます。このような書き込み(実はバッファリング)が繰り返され、バッファがいっぱいになったとき初めて、ディスクへの実際の書き込みが行われるのです(**図2.15**)。

では、バッファがいっぱいにならない時点で、そのファイルへの書き込みがすべて終了してしまった場合はどうなるのでしょう。何回かの実際の書き込みが行われても、最終の書き込みはたいていこの状態だと思われますが、この場合は、そのファイルに対するファイルクローズの操作により、強制的にバッファの内容がディスクへ書き込まれます。

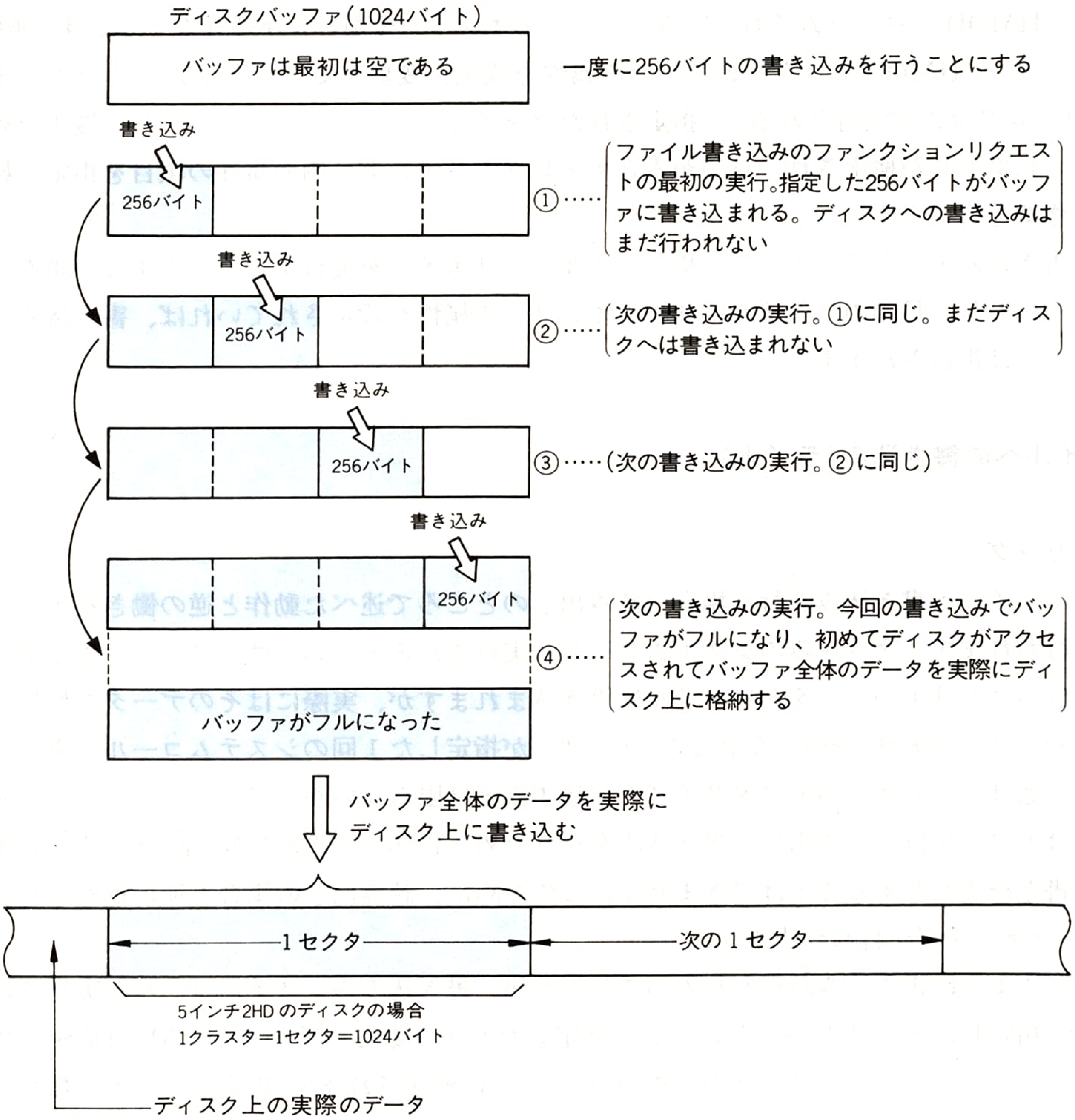
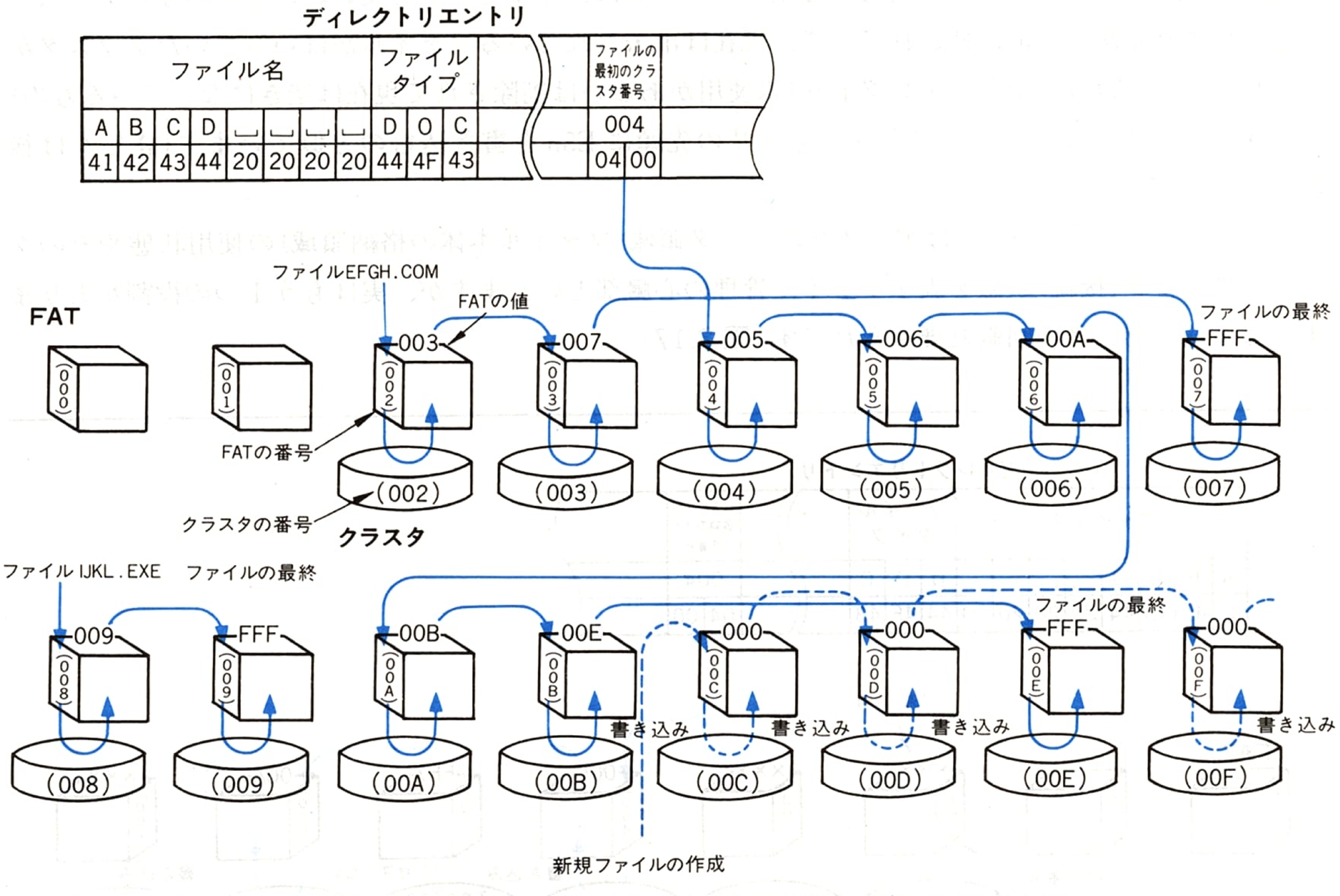

ディスクバッファ（1024バイト）
バッファは最初は空である
一度に256バイトの書き込みを行うことにする
書き込み
256バイト
①……
ファイル書き込みのファンクションリクエストの最初の実行。指定した256バイトがバッファに書き込まれる。ディスクへの書き込みはまだ行われない
書き込み
256バイト
②……
次の書き込みの実行。①に同じ。まだディスクへは書き込まれない
書き込み
256バイト
③……（次の書き込みの実行。②に同じ）
書き込み
256バイト
④……
次の書き込みの実行。今回の書き込みでバッファがフルになり、初めてディスクがアクセスされてバッファ全体のデータを実際にディスク上に格納する
バッファがフルになった
バッファ全体のデータを実際にディスク上に書き込む
1セクタ
次の1セクタ
5インチ2HDのディスクの場合
1クラスタ＝1セクタ＝1024バイト
ディスク上の実際のデータ

図 2.15　ファイル書き込み時のディスク・バッファリングの原理

FAT(File Alocation Table)

ファイルへの書き込みを行っている場合、さきほど述べたディスク・バッファリングのためのバッファがいっぱいになり、いよいよディスクへの書き込みが行われる際には、まずディスク上のどこに書き込めばよいかを決定しなくてはなりません。それを決めるのは FAT の役目ですが、どのようにしてその格納場所を決定するのでしょう。

前項のファイル読み出しの解説で、FAT とは、ファイル本体がディスク上のどこに存在するか、その配置情報を表すテーブルであると述べましたが、ディスク上の使われていない空き領域に対応する FAT には、未使用であることを示す特別な値(000_H)が書き込まれています (図 2.16 参照)。


ディレクトリエントリ
ファイル名
ファイルタイプ
ファイルの最初のクラスタ番号
A B C D ␣ ␣ ␣ ␣ D O C
41 42 43 44 20 20 20 20 44 4F 43
004
04 00
FAT
ファイルEFGH.COM
FATの値
FATの番号
クラスタの番号
クラスタ
ファイルの最終
000 001 002 003 004 005 006 007
003 007 005 006 00A FFF
(002) (003) (004) (005) (006) (007)
ファイル IJKL.EXE
ファイルの最終
008 009 00A 00B 00C 00D 00E 00F
009 FFF 00B 00E 000 000 FFF 000
(008) (009) (00A) (00B) (00C) (00D) (00E) (00F)
ファイルの最終
書き込み
書き込み
書き込み
書き込み
新規ファイルの作成

次に作成される新しいファイルは、FATの値が000_Hの箇所に対応するクラスタに書き込まれる。この図では、このようなクラスタ順に書き込みが行われる。

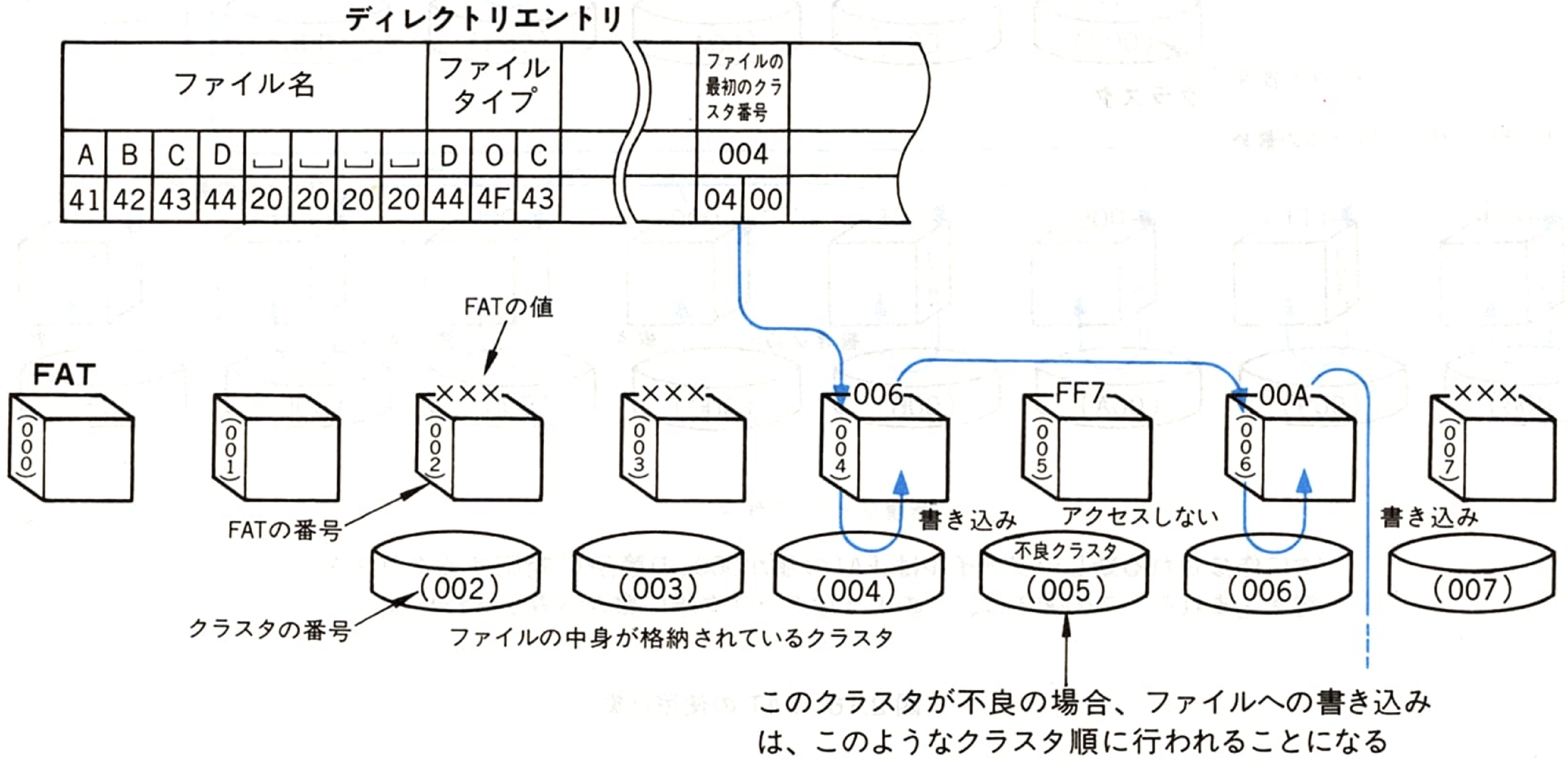
図 2.16　FAT の使用状態

バッファフルになったデータを、ディスク上のどこに書き込むかを決定するには、まずFATを適当なアルゴリズムで検索し、000_HがはいっているFATを捜します。つまり空いているクラスタを捜すのです。そして、空いているクラスタが見つかれば、そこをファイル本体の格納場所として確保し、データの書き込み処理に移ります。見つからない場合は、そのディスクには空き領域がない、つまりディスクフルの状態なので、これ以上の書き込みはできないことになり、ユーザープログラムにはエラー情報が返されます。

現在書き込みを行っているクラスタがいっぱいになった場合は、次の空いているクラスタを同じ方法で捜し出し、見つかったクラスタに続きのデータを書き込んでいきます。これはファイル読み出しの場合とちょうど逆の動作になりますので、前項のファイルの読み出しのところをもう一度参照すれば、どちらもさらによく理解できるでしょう。

000_Hが書かれているFATに対応する空きクラスタは、今まで一度も使われたことのないクラスタかもしれませんが、以前は使われていて、現在は消去されているファイルがはいっていたクラスタかもしれません。これは、空きディレクトリ（未使用かあるいは削除されて現在は空きになっているもの）を示すマークとして、ディレクトリエントリの先頭に$E5_H$を書き込むのと似ています（詳しくは後述）。

さて、以上のようにFATはディスクのデータ領域（ファイル本体の格納領域）の使用状態やそのクラスタのリンク（接続）状態を表すファイル管理の心臓部といえますが、実はもう1つの役割があります。スキップセクタの回避処理がそれです（**図 2.17**）。


ディレクトリエントリ
ファイル名
ファイルタイプ
ファイルの最初のクラスタ番号
A B C D ␣ ␣ ␣ ␣ D O C
41 42 43 44 20 20 20 20 44 4F 43
004
04 00
FATの値
FAT
000
001
×××
002
×××
003
006
004
FF7
005
00A
006
×××
007
FATの番号
書き込み
アクセスしない
不良クラスタ
書き込み
(002)
(003)
(004)
(005)
(006)
(007)
クラスタの番号
ファイルの中身が格納されているクラスタ

このクラスタが不良の場合、ファイルへの書き込みは、このようなクラスタ順に行われることになる

図 2.17　ディスク上のスキップセクタと FAT の関係

取り扱いや使用環境が悪いと、ディスクの磁性面に傷がついたりして、正しくデータを書き込んだり読み出したりできないセクタが生じることがありますが、MS-DOSのファイルシステムには、そのようなスキップセクタが発生しても、そのディスクがまるごと使用できなくなる事態を回避する機能が用意されています。この処理はFORMATコマンドやRECOVERコマンドによって自動的に行われ、スキップセクタを含んでいるクラスタのFATエントリに、不良マーク(クラスタの番号としてはあり得ない特別の値。$FF7_H$)を書き込みます。この処理が行われると、スキップセクタのマークのついたクラスタは、それ以降のアクセスの対象から除外され、ディスクは支障なく使用することができるようになります。もっとも、一度スキップセクタの生じたディスクは、フロッピーディスクであれば、必要なファイルをコピーした上で、すみやかに廃棄したほうが賢明でしょう。

なお、ディレクトリ領域(ルートディレクトリ)とFAT領域は、ディスク上の固定された領域であるため、もしその領域にスキップセクタが発生した場合は、上記の機能の対象にはならず、ディスク全体の致命的なダメージとなります。

FCB(File Control Block)

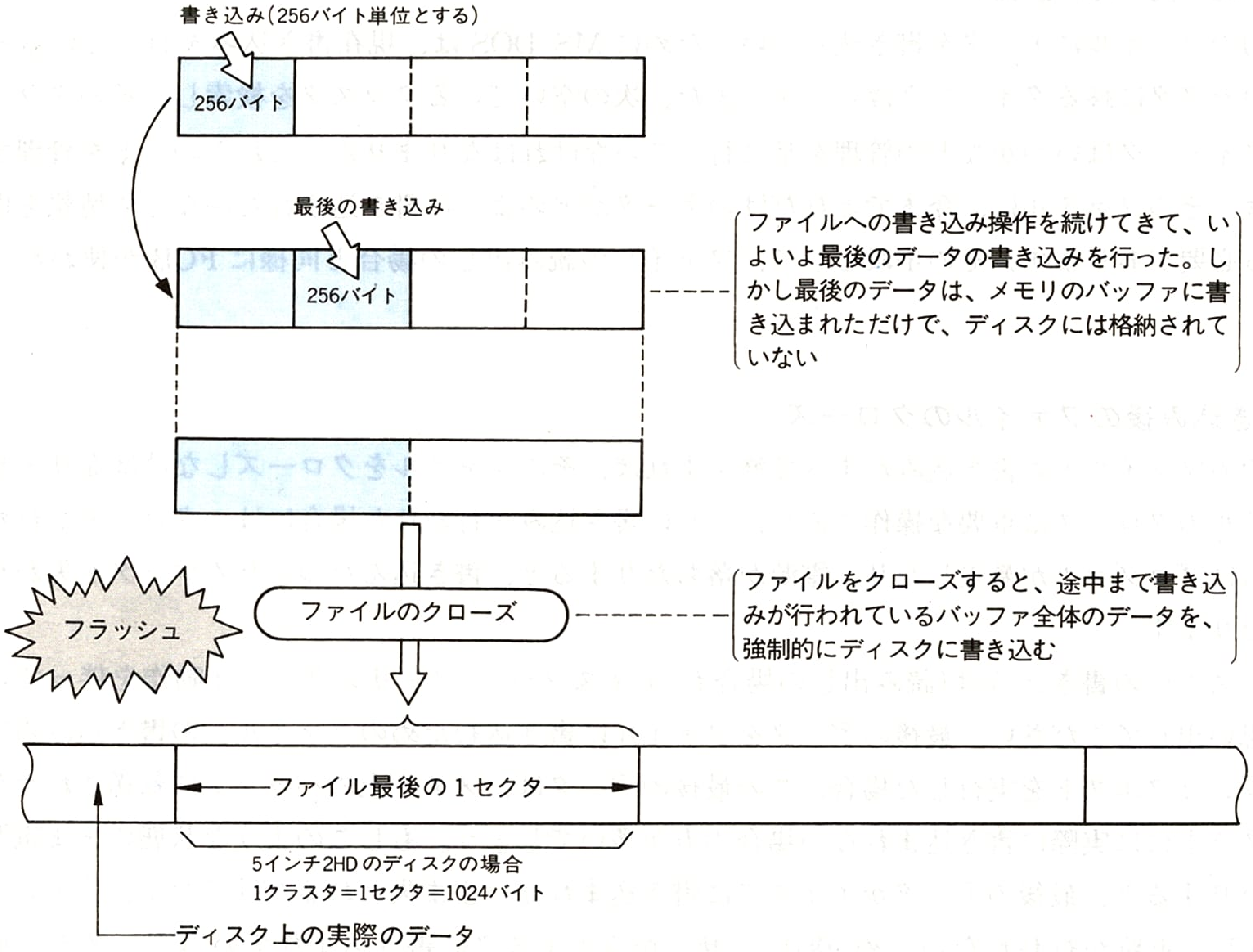
目的のファイルにデータを書き込んでいくためにMS-DOSは、現在書き込みを行っているセクタの次のセクタに移るタイミングはいつか、また、次の空いているクラスタを検索し、そのクラスタに移るタイミングはいつかなどの管理を常に行っていなければなりません。これらのことを管理するためには、そのファイルに、今までどれだけのデータがどのように書き込まれたかなどの情報を保持している必要があります。その手段として、ファイルの読み出しの場合と同様にFCBが使われています。

■ 書き込み後のファイルのクローズ

目的のファイルへの書き込みがすべて終了すれば、そのファイルをクローズしなけばなりません。ファイルのクローズは重要な操作であり、とくに書き込みが行われた場合には、クローズが行われないうちにプログラムが暴走したり、電源が落ちたりすると、書き込んだつもりのデータが失われることになります。

ディスクへの書き込みは(読み出しの場合も)ディスク・バッファリングという動作を伴っていることを思い出してください。最後のデータをファイルに書き込むためのファイルへの書き込みのファンクションリクエストを実行した場合、この最後のデータは、メモリ上のバッファに転送されただけでディスク上には実際に書き込まれない場合の方が多いでしょう。もしこのような状態のまま電源を落としたりすると、最後のデータがディスクに書き込まれないまま失われるだけでなく、ディレクトリやFATの更新が行われないため(後述)、せっかくディスクに書き込まれているデータまで無効になってしまいます。

では、ファイルのクローズによって、どのような処理が行われるのでしょう。ファイルの最終的な書き込みが終了したあと、ファイルクローズのファンクションリクエストを実行することにより、まずバッファに残されているデータがディスクに書き込まれます。これをバッファの**フラッシュ**と呼びます。フラッシュが行われると、次の処理はディレクトリやFATの更新です。ファイルクローズを実行すると、ディレクトリ内のそのファイルのディレクトリエントリに、ファイルのサイズおよびクローズした時点の日付、時間などが書き込まれ、さらにファイルの最初のクラスタを示す番号が書き込まれます。また、そのファイルに対応するFATも、ファイルの中身が格納されているクラスタの正しい接続情報を示すように更新されます。これらの情報は、ファイルオープンのときとは逆に、FCBからコピーされます。ディスクへのすべての書き込み処理が終わると、最後にファイルハンドルおよびそれが指しているFCBの領域が解放され、これでMS-DOSはこのファイルの書き込みに関する管理から完全に離れることになります(**図2.18**)。


書き込み(256バイト単位とする)
256バイト
最後の書き込み
256バイト
ファイルへの書き込み操作を続けてきて、いよいよ最後のデータの書き込みを行った。しかし最後のデータは、メモリのバッファに書き込まれただけで、ディスクには格納されていない
フラッシュ
ファイルのクローズ
ファイルをクローズすると、途中まで書き込みが行われているバッファ全体のデータを、強制的にディスクに書き込む
ファイル最後の1セクタ
5インチ2HDのディスクの場合
1クラスタ=1セクタ=1024バイト
ディスク上の実際のデータ

— 図2.18 —(次ページに続く)

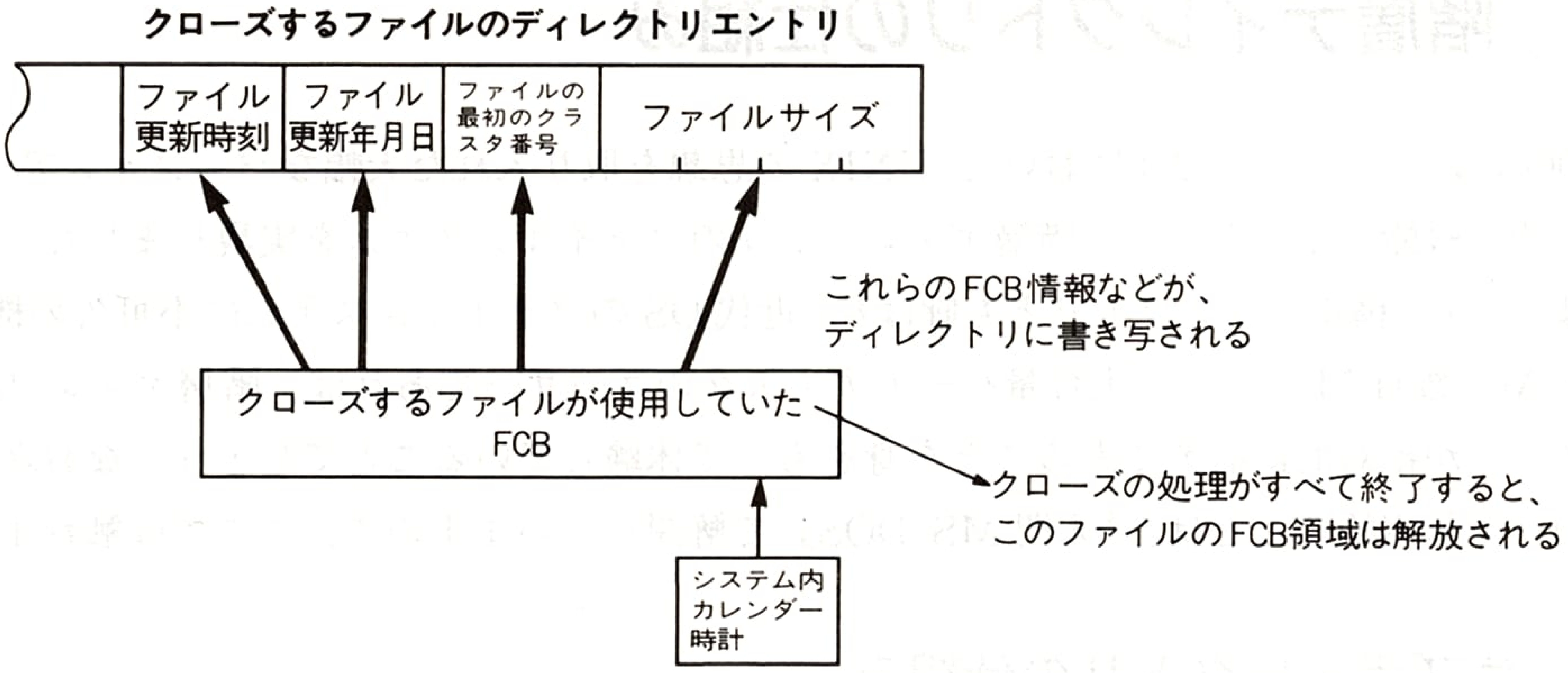
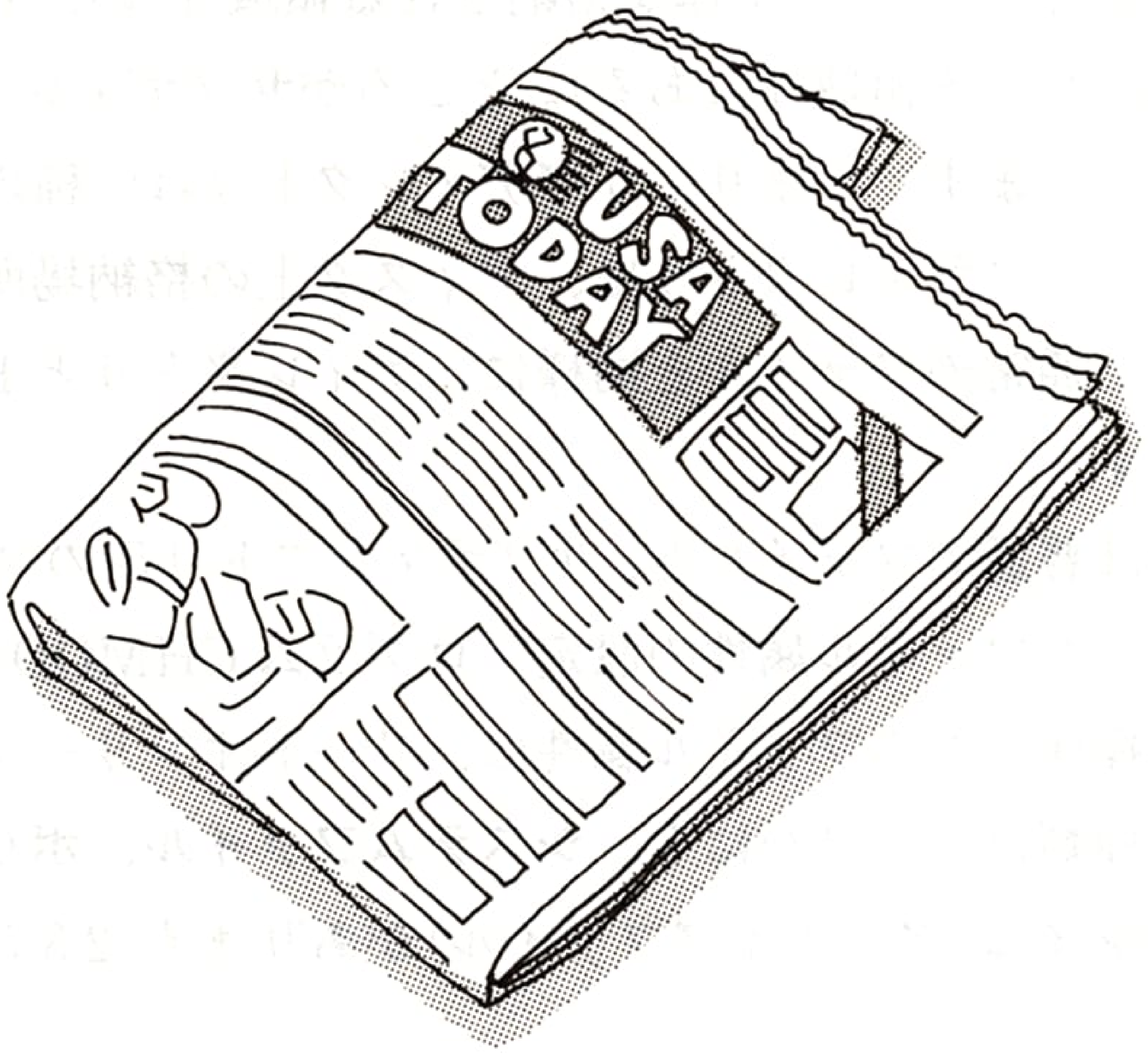

クローズするファイルのディレクトリエントリ
ファイル更新時刻
ファイル更新年月日
ファイルの最初のクラスタ番号
ファイルサイズ
これらのFCB情報などが、ディレクトリに書き写される
クローズするファイルが使用していたFCB
クローズの処理がすべて終了すると、このファイルのFCB領域は解放される
システム内カレンダー時計

図 2.18　書き込み後のクローズで行われる処理

なお、ここで作成したプログラムFWRITEは、ファイルの新規作成機能しかありませんが、このプログラムの一部を変更して、既存ファイルの追加書き込みプログラムを作成する例を4章リスト4.5に示しますので、参考にしてください。



2.2　階層ディレクトリの仕組み

MS-DOSは、バージョン2.0において、UNIXの思想を取り入れた大幅なバージョンアップを行い、その代表的な機能の１つとして、階層ディレクトリのファイルシステムを実現しました。階層ディレクトリは、ツリー構造ディレクトリとも呼ばれ、近代OSのファイルシステムに不可欠の機能です。とくに数十M～数百Mバイトの大容量ハードディスクのユーザーであれば、階層ディレクトリのファイルシステムが絶対的に必要であることを身をもって体験していることでしょう。なお階層ディレクトリについて基本的なことは、『入門MS-DOS』で解説していますので、ここでは触れません。

2.2.1　サブディレクトリの仕組み

階層ディレクトリは、ツリー(木)構造とも呼ばれているように、ルートディレクトリをもとにして、そこにいくつかのサブディレクトリが存在し、そのそれぞれのサブディレクトリにも、さらにサブディレクトリが存在する…、というような階層を際限なく重ねることができる構造になっています。この構造を利用して、ユーザーは多くの種類のファイルを、階層的に整理することができるのです。その構造を「木」にたとえれば、根に直結した幹が**ルートディレクトリ**、枝はすべて**サブディレクトリ**、幹や枝に生えている葉はそれぞれのディレクトリ内の**ファイル**ということになります。

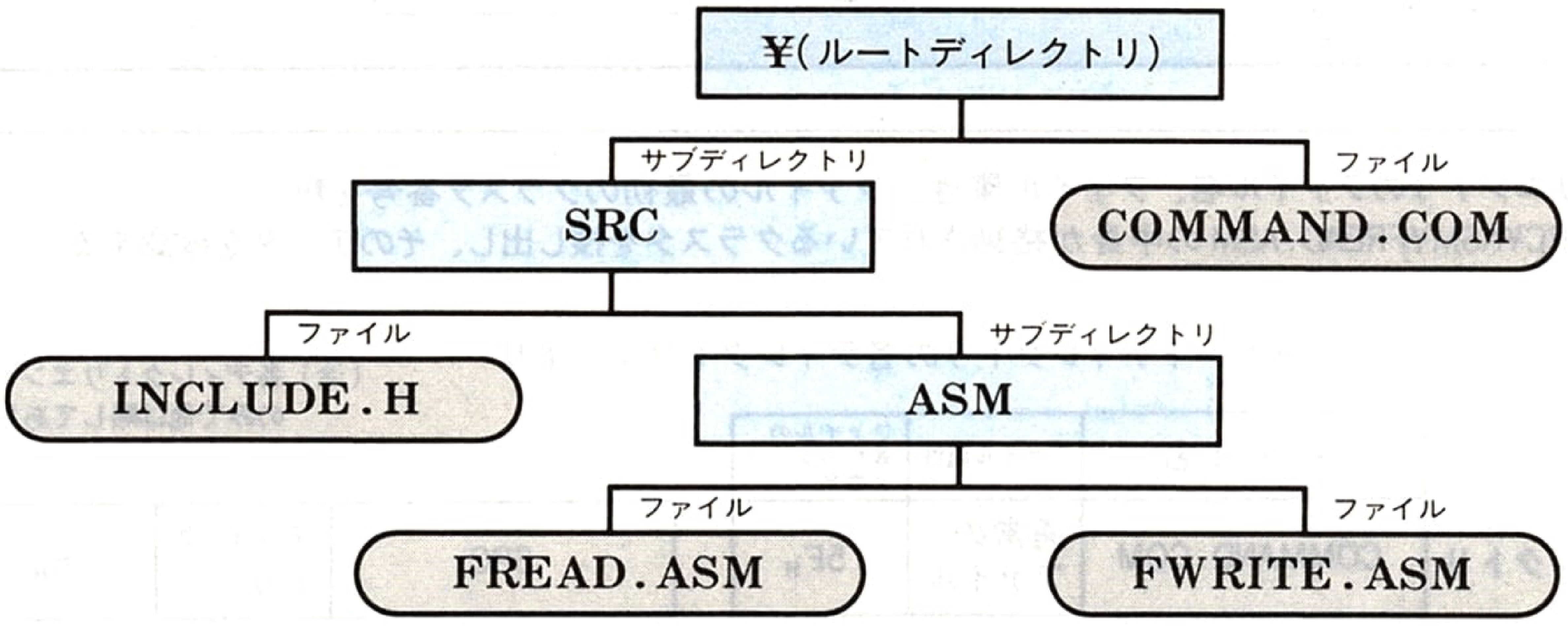
さて、ここではサブディレクトリについて考えてみましょう。まず、サブディレクトリがどのような仕組みで実現されているかについて解説します。

前節では、ディレクトリの仕組みや役割について解説してきましたが、ディレクトリがディスク上のどのような場所に格納されているかについては触れませんでした。実はルートディレクトリは、データ領域外(ファイル本体が格納される領域外)の、ある決まった場所に格納されています(FAT領域も同様にデータ領域外にある)。ところがサブディレクトリはデータ領域の中にあり、ファイル本体と同居しています。つまり、サブディレクトリは一種の「ファイル」として格納されているのです。このようにサブディレクトリは、ディスク上の格納場所が固定されているわけではなく、これまで解説してきた通常のファイルと同様に、ディレクトリとFATによってその格納場所が管理されることになります。

では普通のファイルと、サブディレクトリ用のファイルとはどこが違うのでしょう。ここで、1章で作成したファイル属性の設定プログラムCHMODのことを思い出してください。CHMODプログラムで操作できるファイル属性は、リードオンリーファイルと隠しファイルのみでしたが、ファイル属性の種類はこのほかにも、**システムファイル**、**ボリュームラベル**、**サブディレクトリ**、**未バックアップファイル**(アーカイブファイル)があります(2.3.7の表2.2参照)。

前節で述べましたが、これらのファイル属性を表す情報はそれぞれのディレクトリエントリに書かれており、これがサブディレクトリを表す属性であれば、そのディレクトリエントリが指すのは通常のファイルではなく、サブディレクトリということになります。ですから、そのディレクトリエントリには、ファイル名の代わりにサブディレクトリ名が入れられ、クラスタ番号はファイルの本体が格納されているクラスタを指す代わりに、サブディレクトリの本体が格納されているクラスタを指しています。

では、図2.19に示すような階層ディレクトリについて考えてみましょう。このディレクトリの階層構造は、まず、ルートディレクトリに、ファイルCOMMAND.COMとサブディレクトリSRCがあり、そのサブディレクトリ¥SRC上(図の位置では下)には、ファイルINCLUDE.Hとサブディレクトリ ASMがあります。さらに、そのサブディレクトリ¥SRC¥ASM上には2つのファイルFREAD.ASMとFWRITE.ASMがあるというものです。



¥(ルートディレクトリ)
サブディレクトリ
ファイル
SRC
COMMAND.COM
ファイル
サブディレクトリ
INCLUDE.H
ASM
ファイル
ファイル
FREAD.ASM
FWRITE.ASM

図2.19　例として考える階層構造のディレクトリ

このような階層構造の場合、たとえばそのルートディレクトリから、¥SRC¥ASM¥FREAD.ASMというファイルにたどり着くには、システム内部でどのようなことが行われているのでしょうか。以下に、そのときの動きを、順を追って示します。また、階層構造におけるルートディレクトリ上のサブディレクトリ用エントリと、データ領域内のサブディレクトリ本体の内部の状態を図2.20に示しますので参照してください。

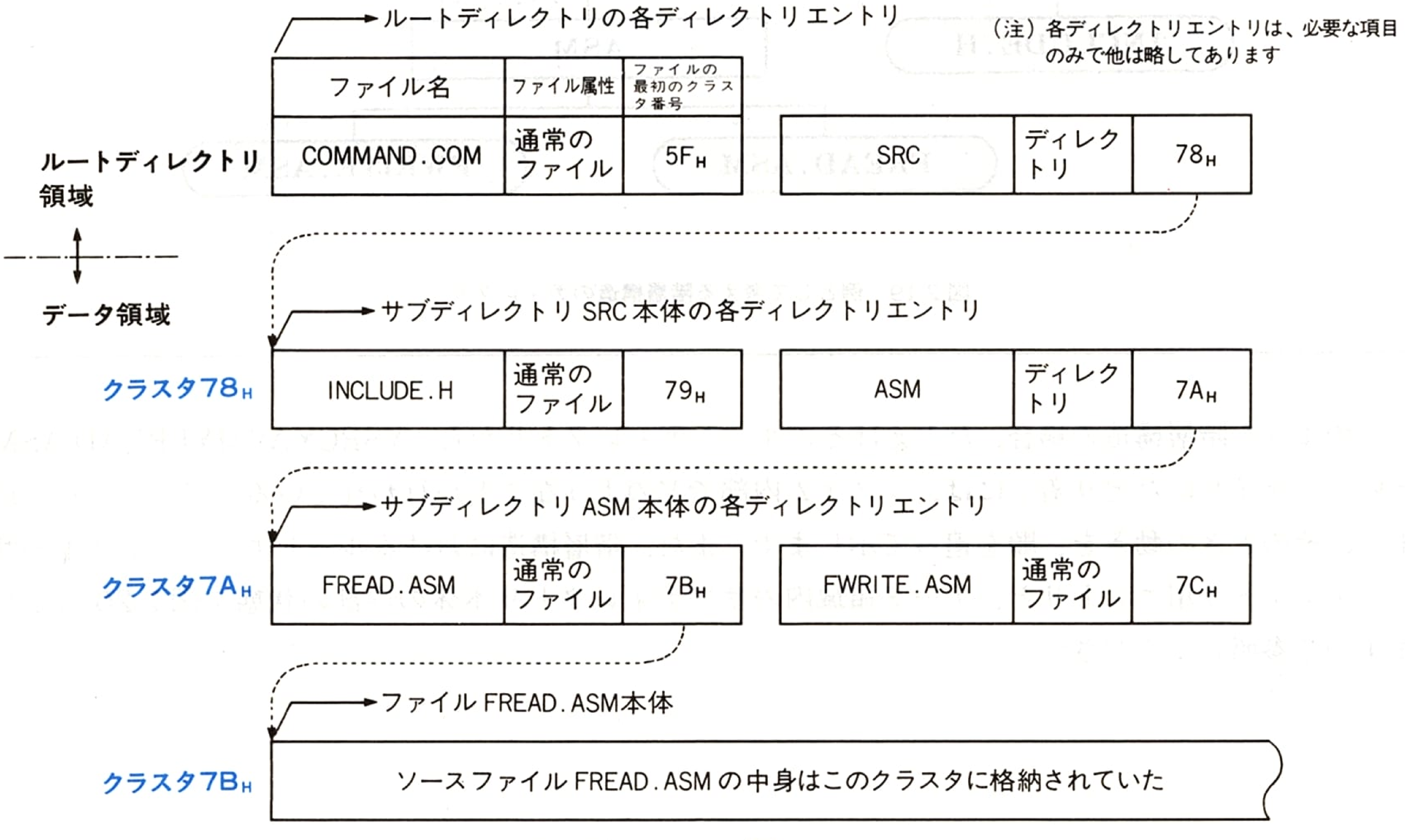
ファイル名¥SRC¥ASM¥FREAD.ASMが与えられると、まずルートディレクトリを検索し、ファイル属性が「ディレクトリ」であり、かつファイル名（この場合はサブディレクトリ名）が「SRC」であるディレクトリエントリを見つけ出す

見つかればそのサブディレクトリ¥SRCの本体はデータ領域に存在しているので、まずそのクラスタ番号が指し示すファイルの本体（この場合はサブディレクトリ¥SRCの本体）を見て、その中からファイル属性が「ディレクトリ」であり、かつファイル名（この場合はサブディレクトリ名）が「ASM」であるディレクトリエントリを見つけ出す

見つかれば、サブディレクトリ¥SRC¥ASMの本体はデータ領域に存在しているので、さきと同様に、そのクラスタ番号が指し示すファイルの本体（サブディレクトリ¥SRC¥ASMの本体）を見て、その中からファイル属性が「ファイル」であり、ファイル名が「FREAD.ASM」であるディレクトリエントリを見つけ出す

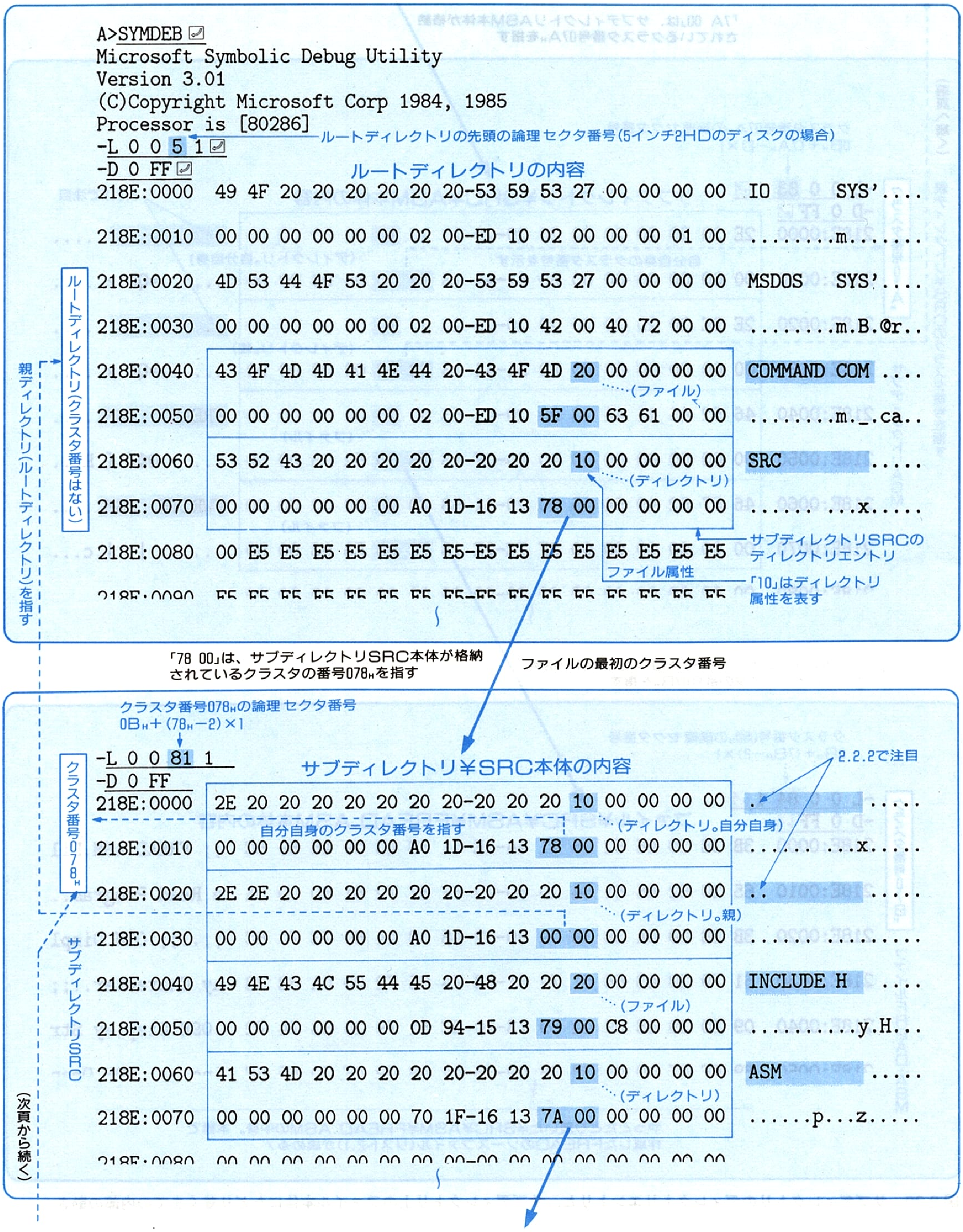
見つかれば目的のファイル¥SRC¥ASM¥FREAD.ASMの本体はデータ領域に存在しているので、そのクラスタ番号が指し示すクラスタが、目的のファイル本体の最初のデータが格納されている場所である

ディレクトリエントリのファイル名、ファイル属性、ファイルの最初のクラスタ番号を頼りに、ファイル¥SRC¥ASM¥FREAD.ASMの中身が格納されているクラスタを捜し出し、そのデータを確認する


ルートディレクトリの各ディレクトリエントリ
（注）各ディレクトリエントリは、必要な項目のみで他は略してあります
ファイル名
ファイル属性
ファイルの最初のクラスタ番号
ルートディレクトリ領域
データ領域
COMMAND.COM
通常のファイル
5FH
SRC
ディレクトリ
78H
サブディレクトリSRC本体の各ディレクトリエントリ
クラスタ78H
INCLUDE.H
通常のファイル
79H
ASM
ディレクトリ
7AH
サブディレクトリASM本体の各ディレクトリエントリ
クラスタ7AH
FREAD.ASM
通常のファイル
7BH
FWRITE.ASM
通常のファイル
7CH
ファイルFREAD.ASM本体
クラスタ7BH
ソースファイルFREAD.ASMの中身はこのクラスタに格納されていた

この追跡作業を実際に行ってみる ⇩

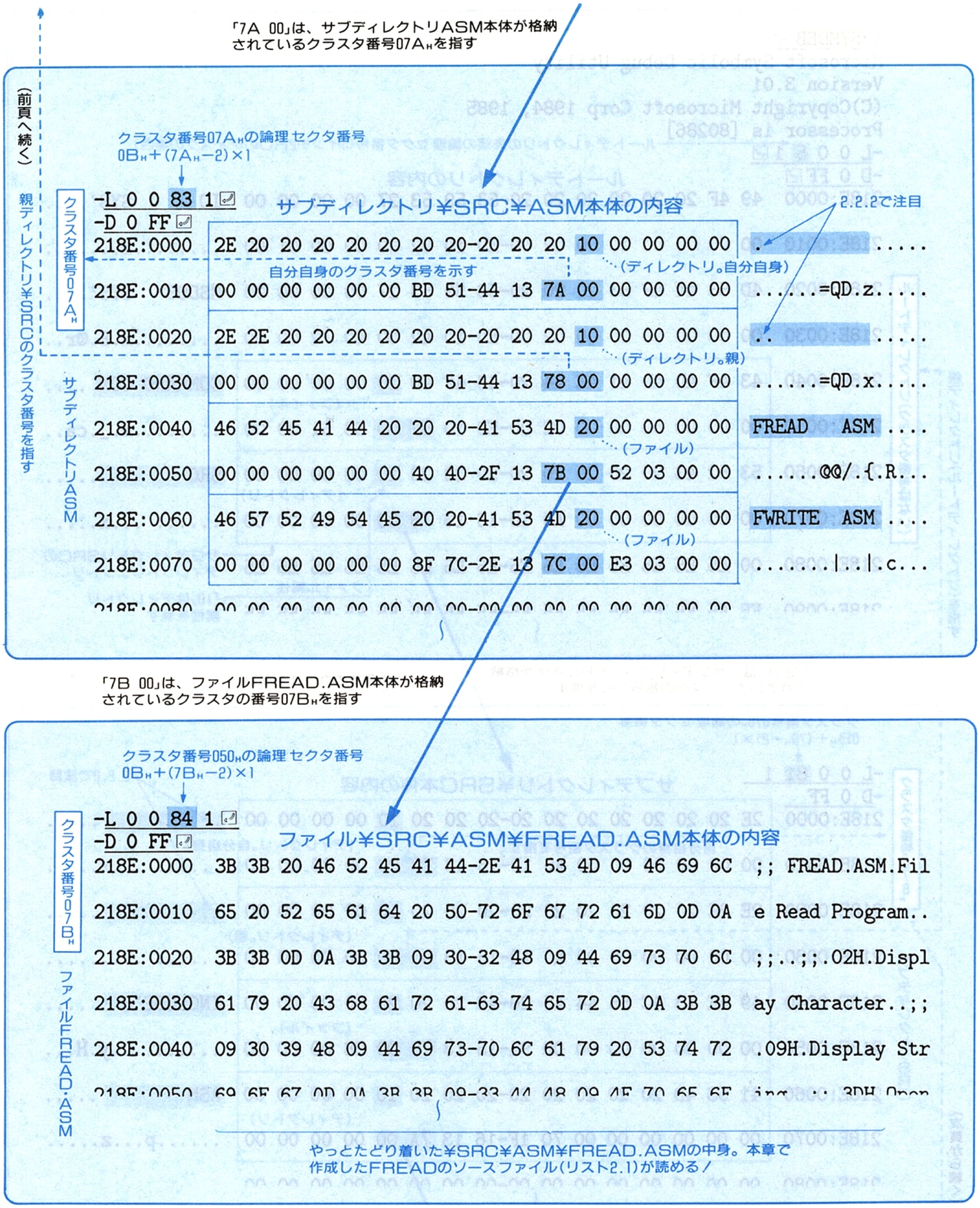
— 図2.20 —（次ページ以下に続く）



```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....

218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......

218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....

218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..

218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....

218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..

218E:0060  53 52 43 20 20 20 20 20-20 20 20 10 00 00 00 00  SRC        .....

218E:0070  00 00 00 00 00 00 A0 1D-16 13 78 00 00 00 00 00  ...... ...x.....

218E:0080  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5

218E:0090  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5
```

```
-L 0 0 81 1
-D 0 FF
218E:0000  2E 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  .          .....

218E:0010  00 00 00 00 00 00 A0 1D-16 13 78 00 00 00 00 00  ...... ...x.....

218E:0020  2E 2E 20 20 20 20 20 20-20 20 20 10 00 00 00 00  ..         .....

218E:0030  00 00 00 00 00 00 A0 1D-16 13 00 00 00 00 00 00  ...... .........

218E:0040  49 4E 43 4C 55 44 45 20-48 20 20 20 00 00 00 00  INCLUDE H   ....

218E:0050  00 00 00 00 00 00 0D 94-15 13 79 00 C8 00 00 00  ..........y.H...

218E:0060  41 53 4D 20 20 20 20 20-20 20 20 10 00 00 00 00  ASM        .....

218E:0070  00 00 00 00 00 00 70 1F-16 13 7A 00 00 00 00 00  ......p...z.....

218E:0080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

ルートディレクトリの先頭の論理セクタ番号(5インチ2HDのディスクの場合)
ルートディレクトリの内容
ルートディレクトリ(クラスタ番号はない)
親ディレクトリ(ルートディレクトリ)を指す
(ファイル)
(ディレクトリ)
サブディレクトリSRCのディレクトリエントリ
ファイル属性
「10」はディレクトリ属性を表す
「78 00」は、サブディレクトリSRC本体が格納されているクラスタの番号078Hを指す
ファイルの最初のクラスタ番号
クラスタ番号078Hの論理セクタ番号
0BH＋(78H−2)×1
サブディレクトリ¥SRC本体の内容
2.2.2で注目
自分自身のクラスタ番号を指す
(ディレクトリ。自分自身)
(ディレクトリ。親)
クラスタ番号078H
サブディレクトリSRC
(ファイル)
(ディレクトリ)
(次頁から続く)


「7A 00」は、サブディレクトリASM本体が格納
されているクラスタ番号07A_Hを指す
(前頁へ続く)
親ディレクトリ¥SRCのクラスタ番号を指す
クラスタ番号07A_Hの論理セクタ番号
0B_H+(7A_H−2)×1
クラスタ番号07A_H
サブディレクトリASM
サブディレクトリ¥SRC¥ASM本体の内容
自分自身のクラスタ番号を示す
(ディレクトリ。自分自身)
(ディレクトリ。親)
(ファイル)
(ファイル)
2.2.2で注目
「7B 00」は、ファイルFREAD.ASM本体が格納
されているクラスタの番号07B_Hを指す
クラスタ番号050_Hの論理セクタ番号
0B_H+(7B_H−2)×1
クラスタ番号07B_H
ファイルFREAD.ASM
ファイル¥SRC¥ASM¥FREAD.ASM本体の内容
やっとたどり着いた¥SRC¥ASM¥FREAD.ASMの中身。本章で
作成したFREADのソースファイル(リスト2.1)が読める！

```
-L 0 0 83 1
-D 0 FF
218E:0000  2E 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00   .    ......
218E:0010  00 00 00 00 00 00 BD 51-44 13 7A 00 00 00 00 00   ......=QD.z.....
218E:0020  2E 2E 20 20 20 20 20 20-20 20 20 10 00 00 00 00   ..   ......
218E:0030  00 00 00 00 00 00 BD 51-44 13 78 00 00 00 00 00   ......=QD.x.....
218E:0040  46 52 45 41 44 20 20 20-41 53 4D 20 00 00 00 00   FREAD   ASM ....
218E:0050  00 00 00 00 00 00 40 40-2F 13 7B 00 52 03 00 00   ......@@/.{.R...
218E:0060  46 57 52 49 54 45 20 20-41 53 4D 20 00 00 00 00   FWRITE  ASM ....
218E:0070  00 00 00 00 00 00 8F 7C-2E 13 7C 00 E3 03 00 00   ......|..|.c...
218E:0080  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

```
-L 0 0 84 1
-D 0 FF
218E:0000  3B 3B 20 46 52 45 41 44-2E 41 53 4D 09 46 69 6C   ;; FREAD.ASM.Fil
218E:0010  65 20 52 65 61 64 20 50-72 6F 67 72 61 6D 0D 0A   e Read Program..
218E:0020  3B 3B 0D 0A 3B 3B 09 30-32 48 09 44 69 73 70 6C   ;;..;;.02H.Displ
218E:0030  61 79 20 43 68 61 72 61-63 74 65 72 0D 0A 3B 3B   ay Character..;;
218E:0040  09 30 39 48 09 44 69 73-70 6C 61 79 20 53 74 72   .09H.Display Str
218E:0050  69 6E 67 0D 0A 3B 3B 09-33 44 48 09 4F 70 65 6E   ing..;;.3DH.Open
```

図2.20　サブディレクトリのディレクトリエントリと、サブディレクトリ上のファイル本体にたどり着くまでの内部の動き

2.2.2 「.」と「..」

さて、日常の作業でよく経験することと思いますが、MD(MKDIR)コマンドで新しいディレクトリを作成すると、作成されたディレクトリには「.」および「..」のディレクトリが自動的に作られます。これが自分自身を示すディレクトリ「.」と、その親ディレクトリを示すディレクトリ「..」であることはご承知のことと思いますが、実はこれら2つのディレクトリに対応するディレクトリエントリも、そのサブディレクトリ内に存在しているのです(まあ当然ですが)。

さきの図2.20に示したように、「.」ディレクトリの本体を指すクラスタ番号には、自分自身のディレクトリが格納されているクラスタ番号が書かれており、「..」ディレクトリの本体を指すクラスタ番号には親ディレクトリが格納されているクラスタ番号が書かれています。ただし、親ディレクトリがルートディレクトリである場合には、そのクラスタ番号には、データ領域には存在しない特別な数値(000H)が書かれています。これは、ディスク上のルートディレクトリの格納領域がファイル本体が格納されるデータ領域とは別の独立した場所に存在しているため、データ領域内の各クラスタを指すクラスタ番号では表せないからです。

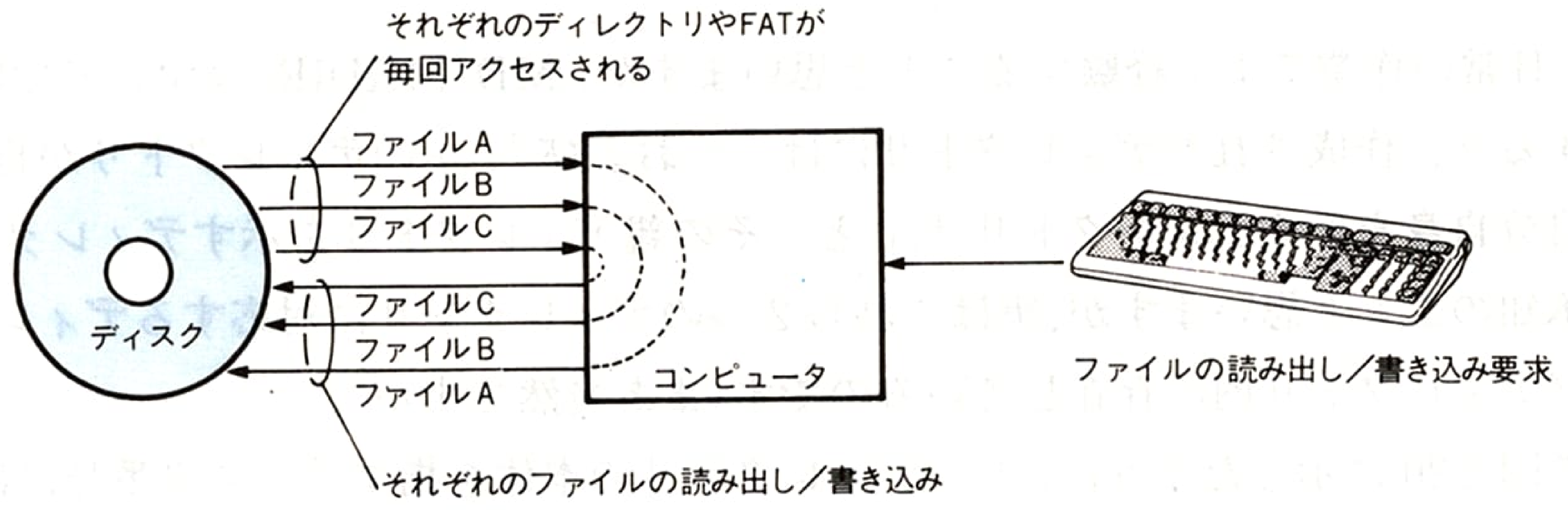
2.2.3 ディレクトリのバッファリング

通常のファイルに対して読み出し／書き込みをする際には、ディスクアクセスを効率よく行うために、ディスク・バッファリングが行われていることは、すでに前節で述べました。そこでも少し触れましたが、ディレクトリやFATに関しても、このバッファリングが行われています。

ディレクトリやFATに書き込まれている情報は、目的のファイルをアクセスするときに必ず参照しなければならない重要なものですが、このディレクトリやFATがディスク上に格納されている場所は、目的のファイル本体が格納されている場所と、必ずしも物理的に近くにあるとは限りません。大きなファイルをアクセスする場合は、途中で何度かFATを参照して、次にアクセスするクラスタを知る必要があります。しかし、そのたびにファイル本体とFAT間の、ディスク上の互いに離れた場所を行ったり来たりしてアクセスするのでは、ハードウェア的に時間がかかりアクセスが遅くなります。そこで、ディレクトリやFATのアクセスに対しても、前節で解説したファイル本体で行われているのと同じバッファリングが行われているのです。

具体的には、何らかのディスクアクセスが実行されると、ディスク上のディレクトリやFATの内容が読み出され、メモリ上のバッファにコピーされます。ファイル本体をアクセスするにはディレクトリやFATの情報を得なければなりませんが、こうしておけばそのたびにディスク上のFATを直接読み出す必要はなく、メモリ上のバッファを参照すればよいことになります。これはメモリ上での作業なので非常に高速で行われ、その結果としてファイルのアクセスが速くなるのです(図2.21)。

■ディレクトリやFATのバッファリングが行われていない場合


それぞれのディレクトリやFATが
毎回アクセスされる
ファイルA
ファイルB
ファイルC
ファイルC
ファイルB
ファイルA
ディスク
コンピュータ
ファイルの読み出し／書き込み要求
それぞれのファイルの読み出し／書き込み

■バッファリングが有効に行われている場合

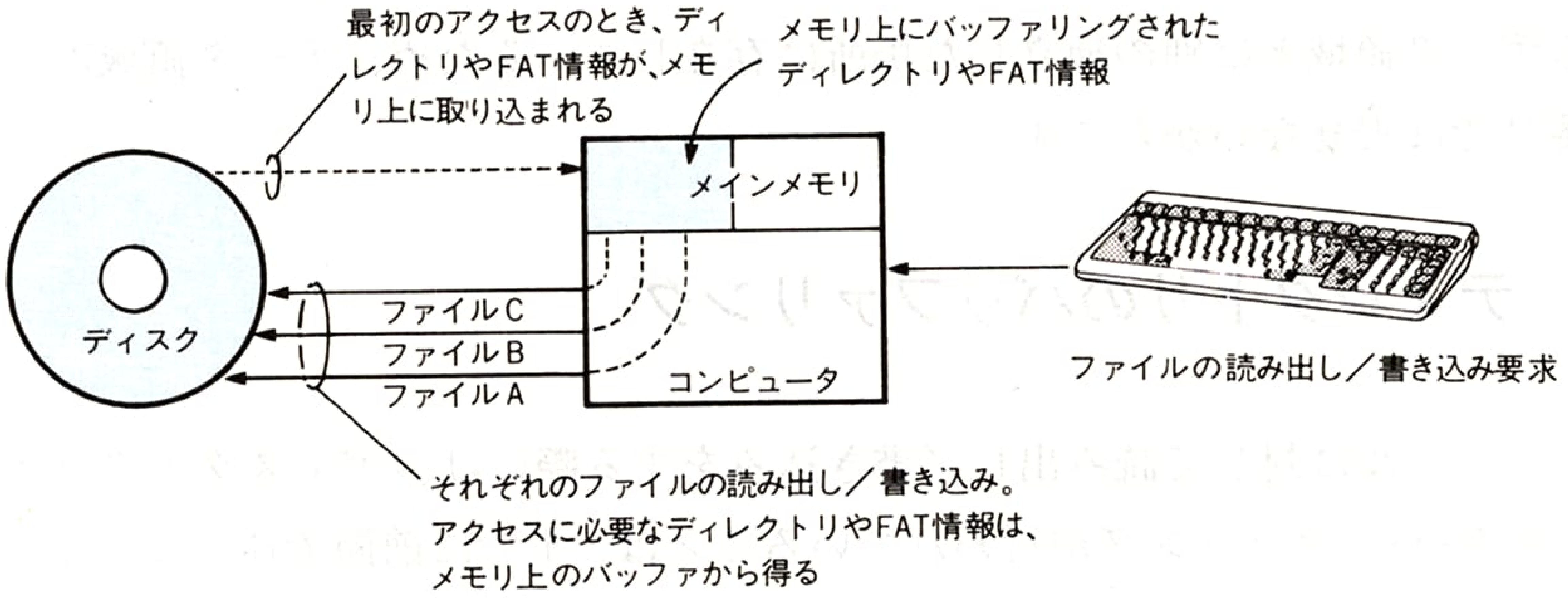

最初のアクセスのとき、ディ
レクトリやFAT情報が、メモ
リ上に取り込まれる
メモリ上にバッファリングされた
ディレクトリやFAT情報
メインメモリ
ディスク
ファイルC
ファイルB
ファイルA
コンピュータ
ファイルの読み出し／書き込み要求
それぞれのファイルの読み出し／書き込み。
アクセスに必要なディレクトリやFAT情報は、
メモリ上のバッファから得る

図2.21　ディレクトリやFATのバッファリング

■アクセス途中でのディスクの交換

みなさんの中には、もし仕事の途中でディスクを交換してしまったらどうなるのか？という疑問を持った人もいるでしょう。つまり、ディレクトリのバッファリングが行われたあと（具体的にはそのディスクを１度でもアクセスしたあと）にディスクを交換した場合です。さきに述べたディレクトリのバッファリングという操作が可能となる前提は、ディスク上のディレクトリやFATの内容を、いったんメモリ上のバッファに読み込んだあとは、そのディスクは絶対に交換されないという条件が必要です。あたりまえのことですが、メモリ上のバッファにコピーされたディレクトリやFATの内容が、ディスク上のものと一致することが保証されていなければ、バッファリングの機能は成立しません。言い換えれば、いつなんどきでも、ディスクが交換された場合には、その事実を確実に検知できる機能がなければ、このバッファリングの機能をサポートすることはできないのです。

もしディスクが交換されたならば、MS-DOSは直ちにその交換の事実を検知して、現在のバッファの内容をすべて破棄し、そのあとにディスクアクセスが行われた場合は、新しいディスクのディレクトリやFATを、バッファに読み込み直すことが必要です(図2.22参照)。

最近のほとんどのディスクドライブは、ドアが開けられたことをハードウェア的に検出できる機能があり、この検出機能を利用して、ディスクが交換されたこと検知しています(この検知方式の場合は、実際にはディスクを交換せず、ドアを開けてそのまま閉じた場合でも、交換されたと判断される)。このようにドアの開閉が検出できるディスクドライブならば、ディレクトリやFATの情報がバッファリングされ、効率のよいファイルアクセスが可能です(5インチ2DDのディスクドライブには、この検出機能がないものもある)。ただし、機種によっては、ディスクの交換を検出できるディスクドライブを使用しているにもかかわらず、その処理を行っていないためにこのバッファリングが有効に働かないという、ディスクアクセスの遅いシステムディスクを提供しているメーカーもあるようですので注意が必要です。なお、RAMディスクやハードディスクでは、一般にメディアを交換することがないので、このバッファリングを行うことが可能です。

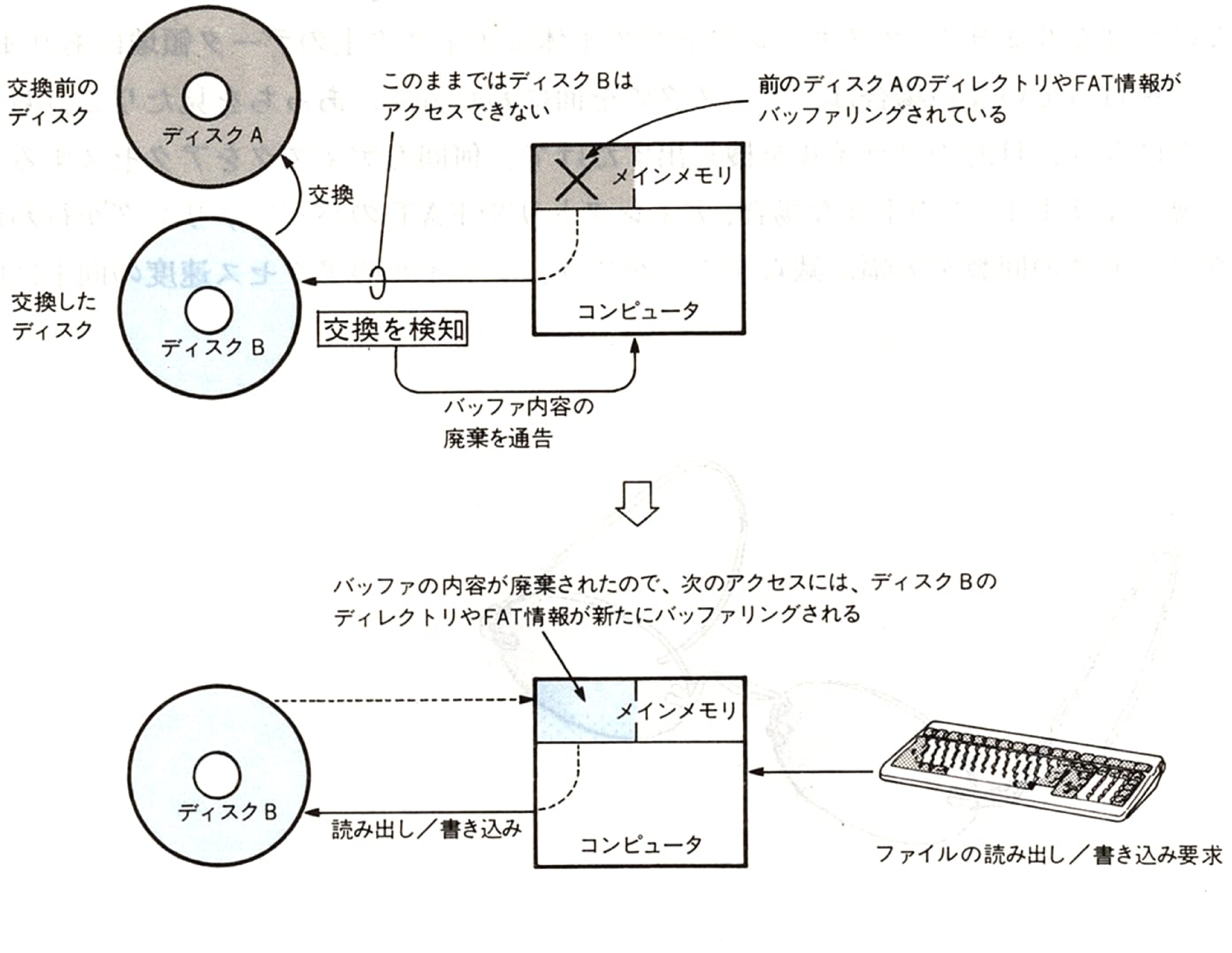
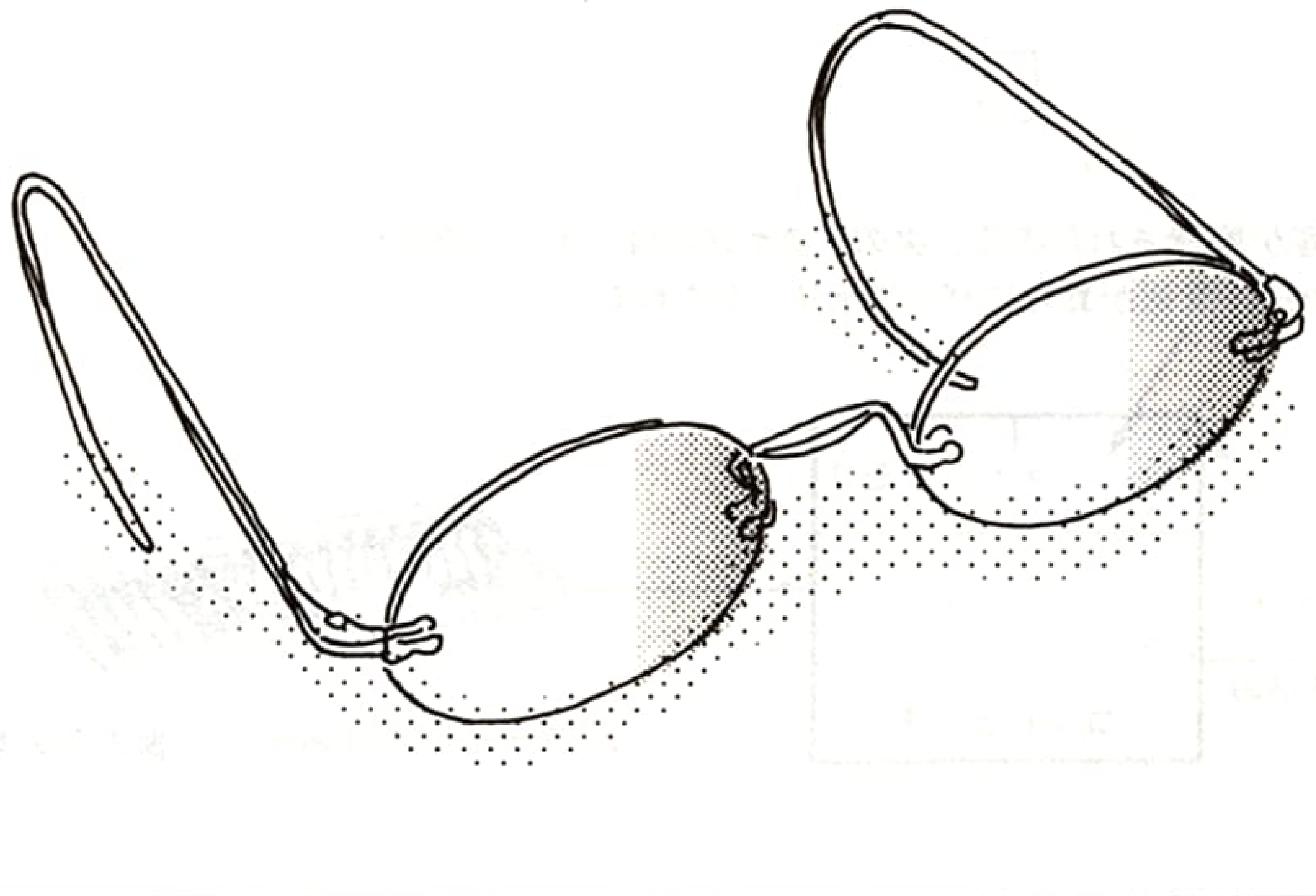

交換前の
ディスク
ディスクA
このままではディスクBは
アクセスできない
前のディスクAのディレクトリやFAT情報が
バッファリングされている
メインメモリ
交換
交換した
ディスク
ディスクB
交換を検知
コンピュータ
バッファ内容の
廃棄を通告
バッファの内容が廃棄されたので、次のアクセスには、ディスクBの
ディレクトリやFAT情報が新たにバッファリングされる
メインメモリ
ディスクB
読み出し／書き込み
コンピュータ
ファイルの読み出し／書き込み要求

図2.22　途中でディスクが交換された場合の処理

■ バッファの数

さて、このようなディレクトリやFATのバッファリングを有効に利用して、高速ディスクアクセスを実現するには、十分な容量のバッファ領域を確保しておかなくてはなりません。MS-DOSには、このバッファ領域を確保するためのコマンドが用意されています。これがCONFIG.SYSファイル内に記述するバッファ数指定行の「BUFFERS＝x」(xは、普通10〜20程度の値を指定するが、ファイルの数が数百〜数千にもなる場合は、40〜50のようにさらに多くする)であり、この行を含めたCONFIG.SYSファイルを、システムディスクのルートディレクトリに置くことにより、指定した容量のバッファがMS-DOSの起動時に確保されます。このCONFIG.SYSファイルによるバッファ容量の指定は、MS-DOSの起動時のみに行われ、途中からバッファの容量を変更することはできません。また、この指定を行わない場合は、MS-DOSバージョン2.xでは最小限のバッファの容量(初期値はBUFFERS＝2の状態)しか確保されず、バッファリングの効果はほとんど発揮されませんので注意が必要です。また、バージョン3.xでは、この値はメインメモリの容量によって変わり、384Kバイトの場合は「5」、512Kバイトの場合は「10」、フル実装640Kバイトの場合は「20」となります(BUFFERS指定については、3章の3.2でも解説している)[*]。

またこのバッファリングは、ディレクトリの階層が深い場合により効果的です。サブディレクトリのそのまたサブディレクトリのそのまた…、のファイルをアクセスするには、そのパスを何段階もたどっていかなければなりません。サブディレクトリの本体はディスク上のデータ領域にありますので、バッファリングを行っていない場合は、ディスクの全面にわたって、あっちを見たりこっちを見たりを繰り返すことになり、目的のファイルを捜し出すだけで、何回もディスクをアクセスするため、きわめて能率が悪くなります。このような場合、ディレクトリやFATのバッファリングが行われていれば、ディスクアクセスの回数を大幅に減らすことができ、ファイルのアクセス速度の向上に劇的な効果があるのです。



---

[*] CONFIG.SYSファイルの作成法や、バッファリングによる効果のテスト法などは、『実用MS-DOS』の「3章 RAMディスク、ディスクキャッシュ、EMS」で解説しているので参照されたい。

2.3 ファイルアクセスのメカニズム

さて、これまでMS-DOSのファイルシステムについて、理論的な面から解説してきましたが、ディスク上に各データがどのように配置されているか(これをディスクフォーマットと呼ぶ)などの物理的な面については触れませんでした。このような物理的なことは、普通はまったく意識する必要はありませんが、ファイルシステムとディスクフォーマットの知識を活用すれば、たとえば誤って消去してしまったファイルを復活させることも不可能ではありません。本節では、ディスク上の各部のデータを実際に目で見ながら、ディスクフォーマットについて解説するとともに、今まで解説してきたファイルシステムを具体的に検証していきましょう。

2.3.1 ディスクの物理フォーマット

まず、ディスク上のデータの物理的な構成について解説しましょう。ディスクは、物理的には**図2.23**のように、同心円状に**シリンダ**、円周方向に**セクタ**と呼ばれる記録の単位があります。両面ディスクの場合には、さらに各シリンダごとに表(サイド0)と裏(サイド1)があり、それぞれを**トラック**と呼びます[1]。ハードディスクの場合には、一般に複数のディスクが積み重ねられていますので、1つのシリンダにいくつかのトラックがあります。シリンダには、ディスクの外周から順に番号が付けられ、またトラックにも、たとえば両面ディスクの場合は表→裏→表→裏の順に、外周から通し番号が付けられます[2]。

セクタは、1つのトラックをディスクの円周方向にいくつかに分割したブロックで、物理的なディスクアクセスの最小単位です。セクタの呼び方は、各セクタを特定できるようにトラックの番号を付けて呼ばれ、たとえば「第2トラックの第8セクタ」のように表現します。

MS-DOSで使用するディスクには、ディスク上のトラックの数や1つのトラック上のセクタの数、それに1つのセクタのサイズ(1セクタは何バイトか)などが、ディスクの種類によって「標準ディスクフォーマット」として決められています(表2.1参照)。

---

[1] ただし、このような呼び方は慣用的なものであり、フロッピーディスクではトラック=シリンダという意味で使うことの方が多いので本書でもその意味で解説する。

[2] フロッピーディスクの場合は、トラック=シリンダの意味で使うことが多いので、通常通し番号を付けて呼んでいない。

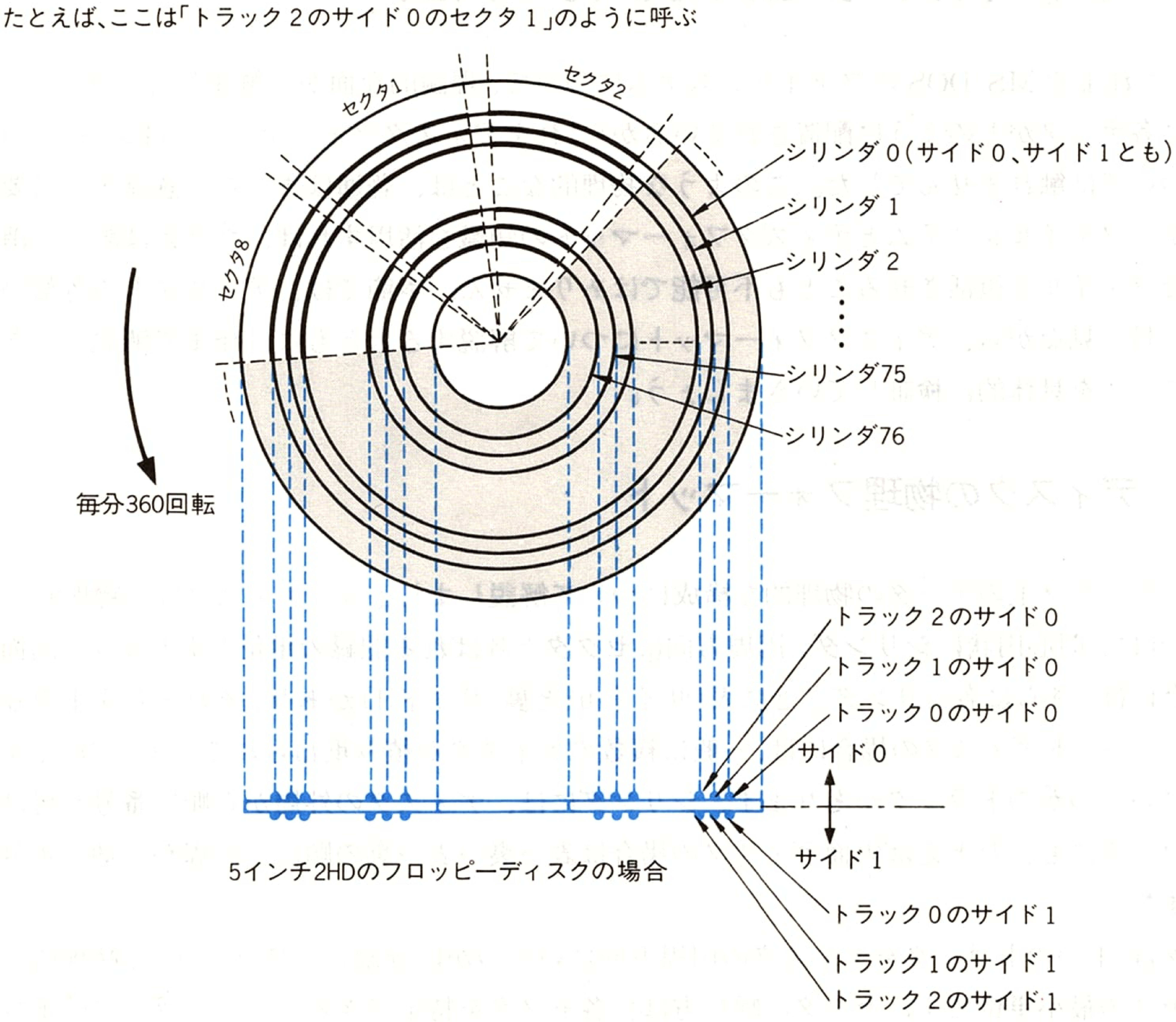

たとえば、ここは「トラック2のサイド0のセクタ1」のように呼ぶ
セクタ1
セクタ2
セクタ8
シリンダ0（サイド0、サイド1とも）
シリンダ1
シリンダ2
シリンダ75
シリンダ76
毎分360回転
トラック2のサイド0
トラック1のサイド0
トラック0のサイド0
サイド0
サイド1
トラック0のサイド1
トラック1のサイド1
トラック2のサイド1
5インチ2HDのフロッピーディスクの場合

図2.23　ディスクのシリンダ、サイド、トラック、セクタの関係

2.3.2　ディスク上の領域

ディスク上には、ファイルの本体を書き込む**データ領域**（**クラスタ領域**とも呼ぶ）のほかに、ファイルの読み出し／書き込みを管理するための**ディレクトリ領域**や**FAT領域**があることは前節で述べましたが、ディスク上にはそのほかに、MS-DOSを起動するためのブートプログラムが書き込まれている**システム予約領域**があります。これら各領域の配置を図2.24に示しましょう。

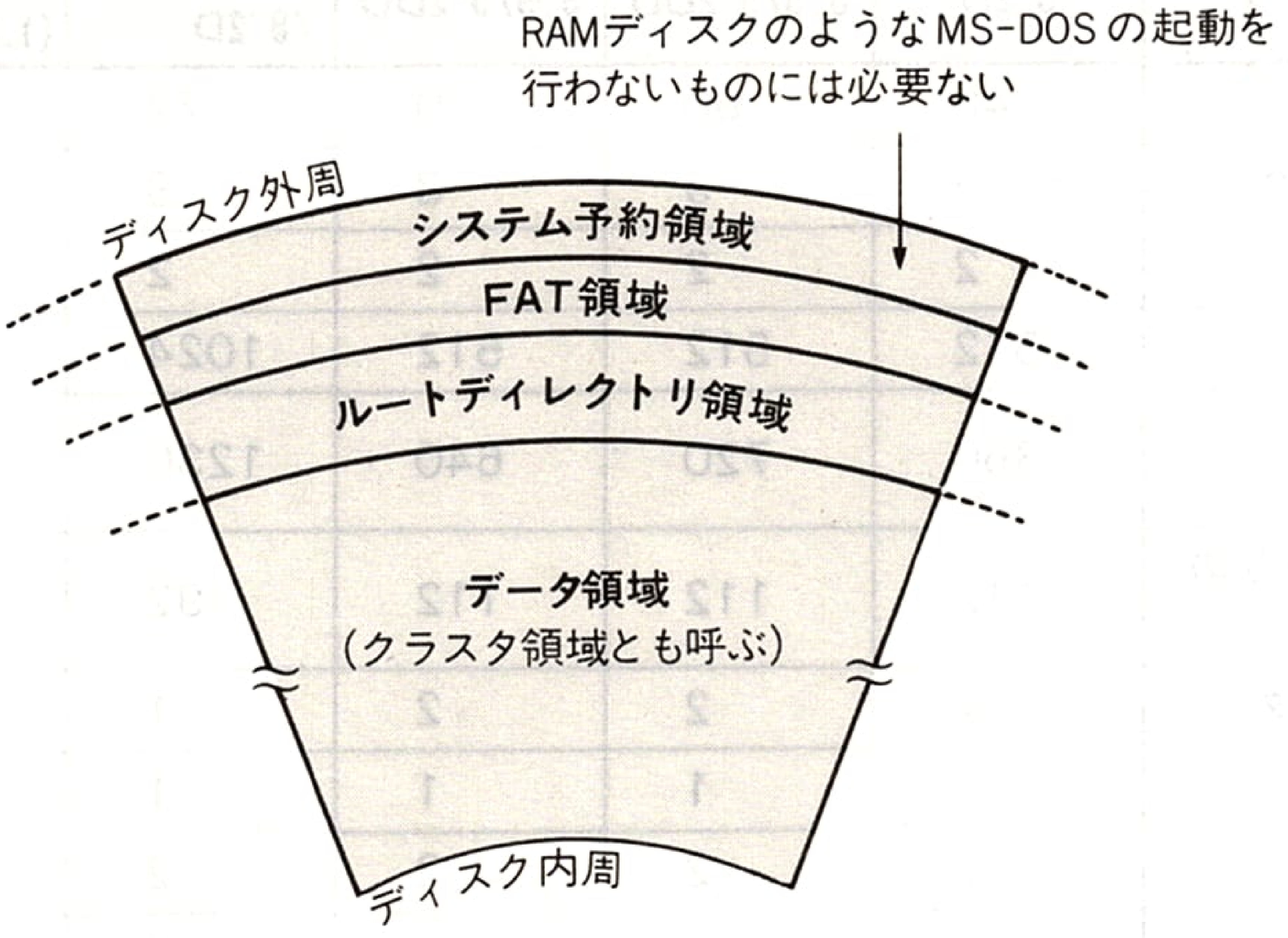

RAMディスクのようなMS-DOSの起動を
行わないものには必要ない
ディスク外周
システム予約領域
FAT領域
ルートディレクトリ領域
データ領域
(クラスタ領域とも呼ぶ)
ディスク内周

図 2.24　ディスク上の各領域の配置

これら4つの領域は、ディスクのトラック番号およびセクタ番号の小さい方から順に割り付けられていますが、それぞれの領域の大きさはディスクの種類によって決められています。MS-DOSでは、これらの領域の配置や大きさがディスクの種類で、統一されていることによって各機種間でのフロッピーディスクのデータの互換性が保たれており、これがMS-DOSの優れた特徴の1つになっています。

さてディレクトリ領域は、ルートディレクトリを格納する領域であり、その大きさはディスクの種類によって決められています。つまり、ルートディレクトリに登録可能なファイルやサブディレクトリの数には、次項の表2.1に示すような制限があるのです。ところが、サブディレクトリは一般的なファイルと同じ扱いでデータ領域に作成されるので、登録できるファイルやディレクトリの数はデータ領域の大きさの範囲内で無制限となります。

また、サブディレクトリがデータ領域のどこに置かれるかは、一般のファイル本体がどこに置かれるのかが不定であるのと同様に一定していません。その位置はファイルが作成された順序や、消去されたファイルやディレクトリの存在などの状況によります。

2.3.3　MS-DOS標準ディスクフォーマット

前項でも述べましたが、ディスク上の**システム予約領域**、**FAT領域**、**ディレクトリ領域**、**データ領域**の4つの領域のそれぞれの大きさや1クラスタが何セクタであるかなどは、ディスクの種類によって決められています。**表2.1**は、MS-DOSで使用する代表的な種類のフロッピーディスクの標準フォーマットの一覧表です。

ディスクのタイプ	5"2D	3.5/5"2DD	3.5/5"2DD	3.5/5"2HD /8"2D	3.5"2HD (1.44Mバイト)	3.5/5"2HC
トラック数	40	80	80	77	80	80
セクタ/1トラック	9	9	8	8	18	15
サイド数	2	2	2	2	2	2
セクタ長	512	512	512	1024	512	512
ディスク全容量 (単位:Kバイト)	360	720	640	1230	1440	1200
ルートディレクトリの 最大エントリ数	112	112	112	192	224	224
セクタ/1クラスタ	2	2	2	1	1	1
予約セクタ数	1	1	1	1	1	1
FAT数	2	2	2	2	2	2
セクタ/1FAT	2	3	2	2	9	7
FAT ID	FD_H	$\mathrm{F9}_H$	FB_H	FE_H	$\mathrm{F0}_H$	$\mathrm{F9}_H$

表2.1　MS-DOSで使用する代表的なディスクの標準フォーマット

表2.1の**FAT数**というのは、まったく同じ内容のFATが、この数だけ用意されていることを示しています。これは、FATを格納しているセクタのデータが、何らかの原因で破壊されたりすると致命的なダメージを受けるため、FATの予備が用意されているのです。また**FAT　ID**というのは、それぞれのディスクのディスクフォーマットを識別するためのコードです。これはたとえば、5インチ2DDのディスクに1トラックあたり8セクタと9セクタの2つのフォーマットがあるように、外形などの物理的条件では、フォーマットの種類を識別できないために必要となります。

FAT　IDは、FATの先頭に書き込んであり、MS-DOSが初めてディスクをアクセスする場合は、まず最初にこのFAT IDを参照して、ディスクの種類を識別します。これによって、使用されるディスクに対応したディレクトリやFATを管理する準備が行われ、本格的にファイルのアクセスが開始されるのです。これらの準備は、接続されているディスクドライブごとに行われますので、たとえば5インチ2DDのディスクであれば、8セクタと9セクタの2種類のフォーマットのディスクを同時にアクセスすることも可能となります。

このようにFATは、ディスクアクセスの最も基本的な情報であるディスクの種類を識別するコードを持っています。この識別コードを読むまでは、MS-DOSにはディスクの種類がわかっていないので、この識別コードだけは、どのような種類のどのようなフォーマットのディスクでも機械的に読めなければなりません。そのために、FATの位置はどのようなディスクでも同じ位置に固定されているのです。

2.3.4　デバッガによる直接ディスクアクセス

これまでに、ディスクフォーマットの基礎的な解説を行いましたので、ここではそれをもとに、ディスクの中身を実際にのぞいてみましょう。

MS-DOSには、ファイルの管理とは無関係に、指定したセクタを直接読み出したり書き込んだりすることのできる機能(4.3節の「システムコール」参照)があります。MS-DOSシステムディスクに付属しているデバッガ(MS-DOSバージョン2.*x*では**DEBUG**、バージョン3.*x*では**SYMDEB**)には、この機能を利用したセクタ読み出しのコマンドL(Load)、およびセクタ書き込みのコマンドW(Write)が用意されており、このコマンドを使えば任意のセクタを自由に読み書きすることができます。

まず、SYMDEB(DEBUGでも同様)のLコマンドを使って、任意のセクタの中身を読み出してみましょう。Lコマンドは、指定したセクタのデータをメモリ上に読み込む(ロードする)コマンドです。読み出そうとするセクタを指定するには、すべてのセクタに通し番号を付けた、Lコマンド独特のセクタ番号を用います。これは、トラック0のセクタ1から始まり、最後のトラックの最後のセクタまで、順番に番号を付けたものです*。そのため、SYMDEBのLコマンドやWコマンドで、任意のセクタを読み出したり書き込んだりするには、このコマンド独特のセクタ番号、つまり論理セクタ番号を、それぞれのディスクの種類別に算出しておかなくてはなりません。表2.1に示したディスクフォーマットの一覧表を参照して、各自が使っているディスクの論理セクタ番号を算出してください。

では、セクタ読み出しの実行例を示しましょう。たとえば、トラック0のセクタ1を読み出すには、**図2.25**のように、デバッガを起動してから、Lコマンドに各パラメータをセットして実行します。与えるパラメータは順に、「読み出したセクタデータを格納するメモリアドレスの先頭値」「読み出すドライブ名」「目的のセクタの論理セクタ番号」「指定したセクタに続けて、連続何セクタ分を読み出すかの値」となります。このLコマンドを実行した際に、ディスクがアクセスされるのを確認してください。

このときのアクセスで、指定したセクタのデータが指定したメモリアドレスに読み込まれていますのでそれを見てみましょう。メモリの内容を見るには、同じSYMDEBのD(Dump)コマンドを実行します。Dコマンドの使い方は、5.6節を参照してください。

---

* MS-DOSでは、各セクタに付けられたこの通し番号のことを論理セクタ番号と呼び、本書でも、この呼び方を採用するが、論理レコード番号と呼ばれることもある。

Lコマンド
読み出したデータを格納するメモリアドレス(通常0にする)
アクセスするドライブ番号(「A：」なら0、「B：」なら1、…)
読み出す論理セクタ番号(この値を変えることによって任意のセクタが読み出せる。0はディスクの最初のセクタ)
連続した何セクタ分を読み出すかの値

```
A>SYMDEB⏎ ……………まずデバッガを起動する
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 0 1⏎
-D 0 FF⏎ ……………Dコマンドで、読み出したデータが格納されているアドレス00H〜FFHをダンプする
218E:0000  EB 1C 90 4E 45 43 20 32-2E 30 30 00 04 01 01 00   k..NEC 2.00.....
218E:0010  02 C0 00 D0 04 FE 02 00-08 00 02 00 00 00 33 C0   .@.P.~........3@
218E:0020  8E D8 8E.C0 8E D0 BC 8A-02 FC BE 0B 00 2E AD 3D   .X.@.P<..|>...-=
218E:0030  80 00 75 38 BE B9 01 B8-00 0A CD 18 B4 0C CD 18   ..u8>9.8..M.4.M.
218E:0040  B4 12 CD 18 0E 1F 33 C0-8E C0 B8 00 A0 26 F6 06   4.M...3@.@8. &v.
218E:0050  01 05 08 74 03 B8 00 E0-8E C0 BF 40 01 AC 0A C0   ...t.8.‘.@?@.,.@
218E:0060  74 04 AA 47 EB F7 B0 06-E6 37 EB FE 3D 00 02 75   t.*Gkw0.f7k~=..u
218E:0070  0E 2E 83 7C 0D 01 74 BC-2E 80 7C 08 FD 74 B5 A0   ...|..t<..|.}t5
218E:0080  84 05 24 F0 BB 00 02 C7-06 02 02 05 01 BA 06 00   ..$p;..G......:..
218E:0090  C7 06 04 02 00 08 C7 06-06 02 00 10 B5 02 2E 80   G.....G.....5...
218E:00A0  7C 08 F9 75 13 3C 90 74-15 C7 06 02 02 06 01 BA   |.yu.<.t.G.....:
218E:00B0  08 00 C7 06 06 02 00 12-A8 80 75 19 EB 2E C7 06   ..G....(.u.k.G.
218E:00C0  02 02 0F 01 BA 01 01 C7-06 04 02 00 02 C7 06 06   ....:..G.....G..
218E:00D0  02 00 1E EB 17 BB 00 04-C7 06 02 02 04 01 C7 06   ...k.;..G.....G.
218E:00E0  04 02 00 14 C7 06 06 02-00 20 B5 03 BD 00 06 B1   ....G.... 5.=..1
218E:00F0  00 A0 84 05 F8 5F 00 1F-0F 1F 51 B9 0B 00 BF F9      h     09  >i
                                    ⟆
```

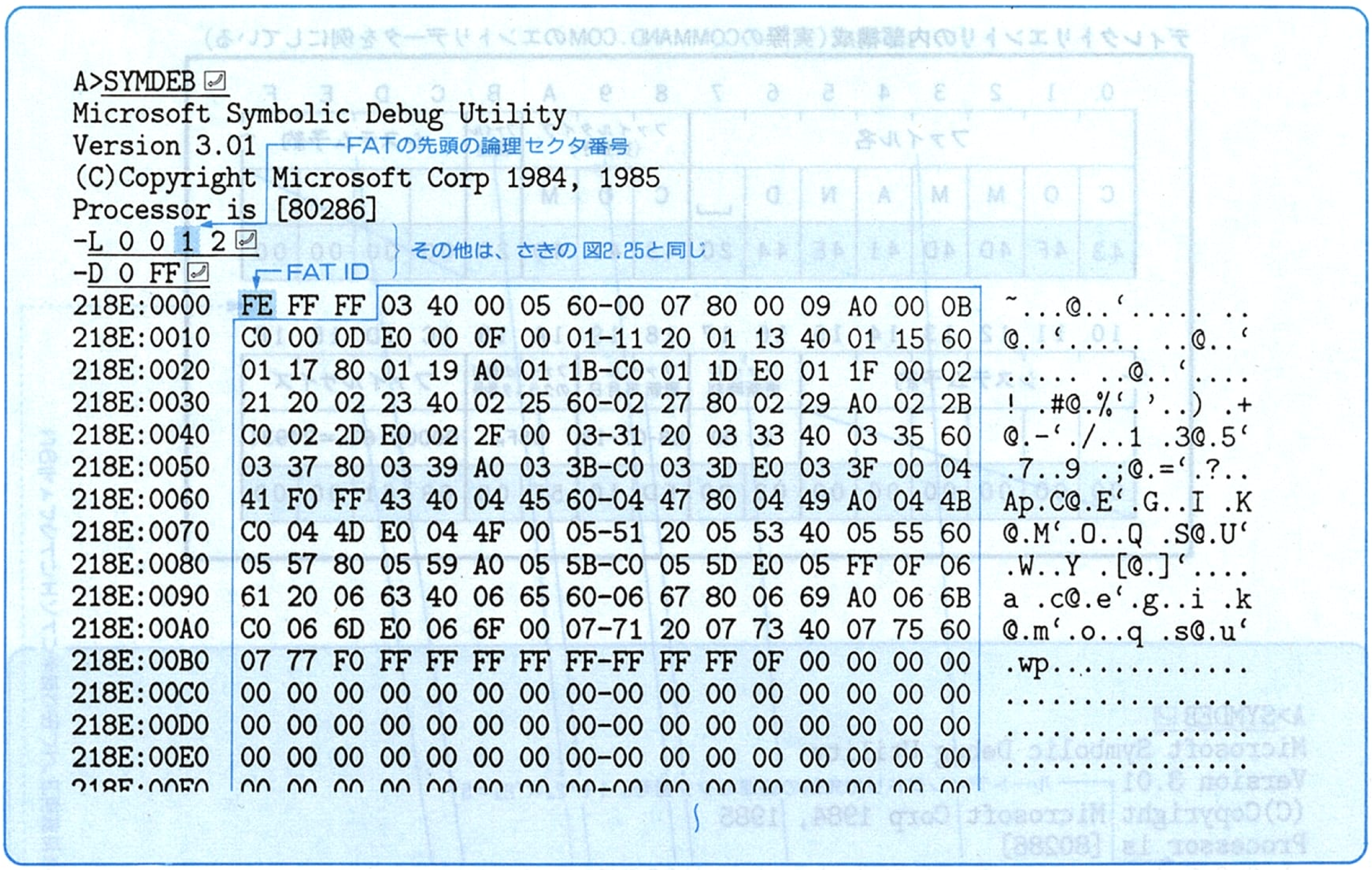
IPL(ブートプログラム)の一部が読み出されている

Aドライブのトラック1のセクタ0(ディスクのいちばん最初のセクタ)を読み出して、そのデータをダンプする実行例

図2.25　任意のセクタのデータを読み出して、メモリ上に読み込まれたセクタのデータをダンプする

2.3.5　FATを見る

目的のセクタを読み出し、そのデータを見ることができるようになりましたので、まず最初にFATの中身を読み出してみましょう。FAT領域の先頭の論理セクタ番号は、システムの予約領域が占めるセクタの数によって異なります。予約領域のセクタ数(予約セクタ)は、ディスクの種類によって異なり、たとえばこれが1の場合は、FAT領域の先頭の論理セクタ番号は1となります(0から始まるので)。各自のディスクの予約セクタの数を表2.1で確認して、FAT領域の先頭の論理セクタ番号を算出してください。なお、ここでの実行例(**図2.26**)は5インチ2HD(8インチ2D、3.5インチ2HDも同じ)のものです。



```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 1 2⏎
-D 0 FF⏎
218E:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B  ~...@..‘...... ..
218E:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60  @..‘..... ..@..‘
218E:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02  ..... ..@..‘....
218E:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B  ! .#@.%‘.’..) .+
218E:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60  @.-‘./..1 .3@.5‘
218E:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04  .7..9 .;@.=‘.?..
218E:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B  Ap.C@.E‘.G..I .K
218E:0070  C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 60  @.M‘.O..Q .S@.U‘
218E:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 FF 0F 06  .W..Y .[@.]‘....
218E:0090  61 20 06 63 40 06 65 60-06 67 80 06 69 A0 06 6B  a .c@.e‘.g..i .k
218E:00A0  C0 06 6D E0 06 6F 00 07-71 20 07 73 40 07 75 60  @.m‘.o..q .s@.u‘
218E:00B0  07 77 F0 FF FF FF FF FF-FF FF FF 0F 00 00 00 00  .wp.............
218E:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
218E:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
218E:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
218E:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

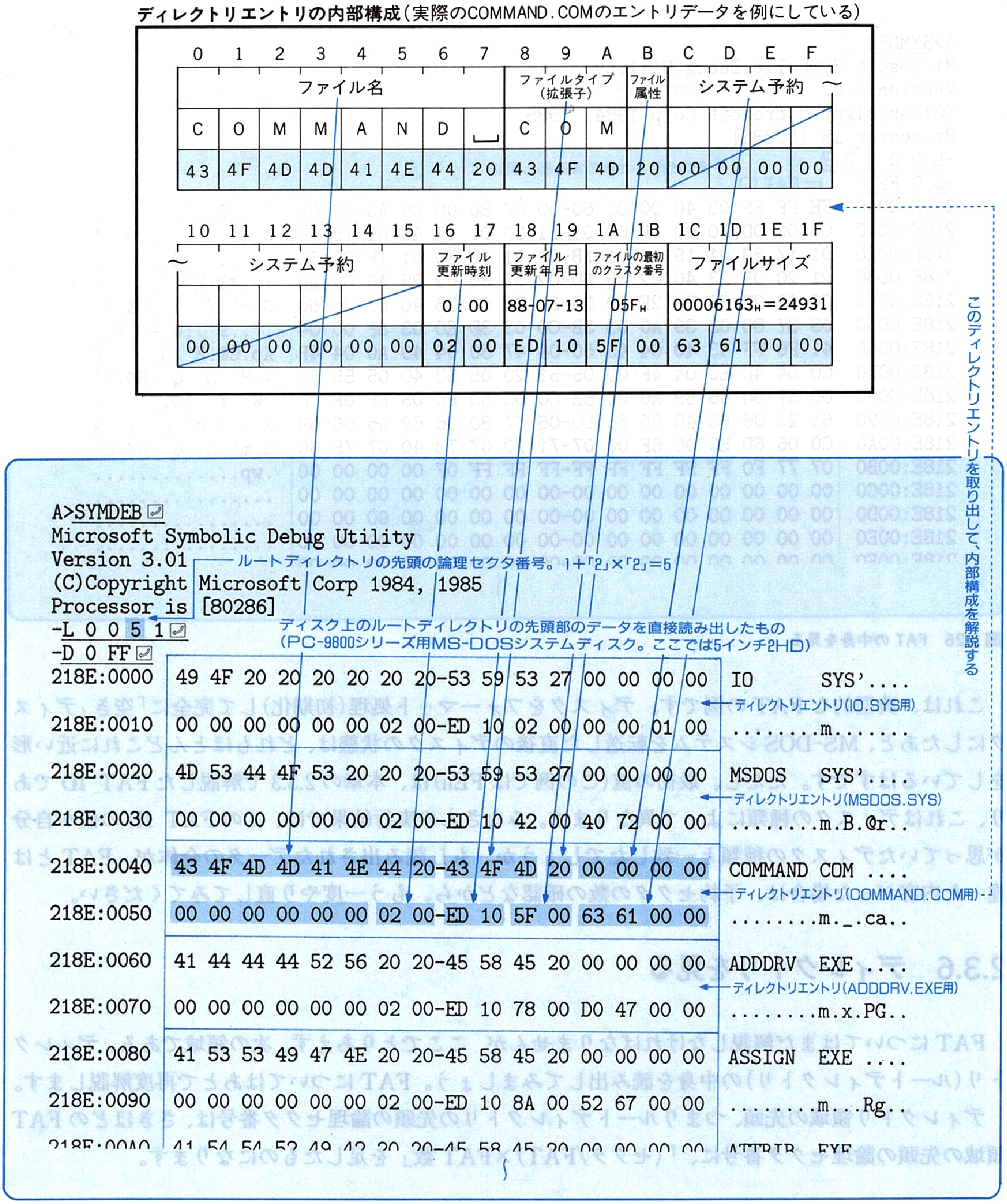
図 2.26　FAT の中身を見る

これは、典型的なFATの例です。ディスクをフォーマット処理(初期化)して完全に「空き」ディスクにしたあと、MS-DOSシステムを転送した直後のディスクの状態は、どれもほとんどこれに近い形をしているはずです。ただし、最初の値(この例ではFE$_H$)は、本章の2.3.3で解説したFAT IDであり、これはディスクの種類によって異なります。みなさんの実行結果では、このFAT IDの値が自分が思っていたディスクの種類と一致したでしょうか。もし読み出されたデータの全体が、FATとは違った内容だった場合は、予約セクタの数の確認などから、もう一度やり直してみてください。

2.3.6　ディレクトリを見る

FATについてはまだ解説しなければなりませんが、ここでとりあえず、次の領域である、ディレクトリ(ルートディレクトリ)の中身を読み出してみましょう。FATについてはあとで再度解説します。

ディレクトリ領域の先頭、つまりルートディレクトリの先頭の論理セクタ番号は、さきほどのFAT領域の先頭の論理セクタ番号に、「(セクタ/FAT)×FAT数」を足したものになります。



ディレクトリエントリの内部構成（実際のCOMMAND.COMのエントリデータを例にしている）

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ファイル名								ファイルタイプ（拡張子）			ファイル属性	システム予約〜			
C	O	M	M	A	N	D	␣	C	O	M					
43	4F	4D	4D	41	4E	44	20	43	4F	4D	20	00	00	00	00

10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
〜システム予約						ファイル更新時刻		ファイル更新年月日		ファイルの最初のクラスタ番号		ファイルサイズ			
						0:00		88-07-13		$05F_H$		00006163_H=24931			
00	00	00	00	00	00	02	00	ED	10	5F	00	63	61	00	00

このディレクトリエントリを取り出して、内部構成を解説する

```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  41 44 44 44 52 56 20 20-45 58 45 20 00 00 00 00  ADDDRV  EXE ....
218E:0070  00 00 00 00 00 00 02 00-ED 10 78 00 D0 47 00 00  ........m.x.PG..
218E:0080  41 53 53 49 47 4E 20 20-45 58 45 20 00 00 00 00  ASSIGN  EXE ....
218E:0090  00 00 00 00 00 00 02 00-ED 10 8A 00 52 67 00 00  ........m...Rg..
218E:00A0  41 54 54 52 49 42 20 20-45 58 45 20 00 00 00 00  ATTRIB  EXE
```

ルートディレクトリの先頭の論理セクタ番号。1+「2」×「2」=5

ディスク上のルートディレクトリの先頭部のデータを直接読み出したもの（PC-9800シリーズ用MS-DOSシステムディスク。ここでは5インチ2HD）

ディレクトリエントリ（IO.SYS用）

ディレクトリエントリ（MSDOS.SYS）

ディレクトリエントリ（COMMAND.COM用）

ディレクトリエントリ（ADDDRV.EXE用）

図2.27　読み出したルートディレクトリのデータと、ディレクトリの各項目との対応

ルートディレクトリの先頭の論理セクタ番号

$$= \underline{\text{FATの先頭の論理セクタ番号}} + \underline{(\text{セクタ/FAT})} \times \underline{\text{FAT数}}$$

予約セクタ数が1なら1, 4なら4　　表2.1より

では、ディレクトリの中身を見てみましょう。それらは、32バイト（ダンプリストの2行分にあたる）で1つのディレクトリエントリを構成しており、これらのデータは、ディレクトリの各項目に**図 2.27**のように対応しています。

2.3.7　ファイルの属性（アトリビュート）を見る

ファイルの属性は、ディレクトリエントリ内に見ることができます。属性の種類はそれぞれ**表 2.2**に示した値で表されています。

属性名称	性　質	属性値	属性値のビットパターン
リードオンリーファイル	消去や更新が禁止されている	01H	00000001
隠しファイル	DIR コマンドでは表示されず、DEL コマンドも無効	02H	00000010
システムファイル	MS-DOS のシステムファイル。リードオンリーでかつ隠しファイル	04H	00000100
ボリューム名	ボリュームラベルであることを示す	08H	00001000
ディレクトリ	サブディレクトリであることを示す	10H	00010000
未バックアップファイル（アーカイブファイル）	MS-DOS バージョン 3.*x* の BACKUP コマンドなどで利用される。通常のファイルはこの属性を持つ	20H	00100000

表 2.2　ファイル属性の種類とそれに対応する値

属性の種類は、それぞれ1バイトの値で表され、その各ビットが属性の種類に対応しています。

IO.SYS や MSDOS.SYS は、システムファイルですので、その属性の値は、システムファイルの 04H であると思われますが、実際には 27H となっています。これは、リードオンリーと隠しファイルとシステムファイルの3つの属性に、さらに未バックアップファイル（アーカイブファイル）*の属性を足した値です。つまり、システムファイルは実はリードオンリーファイルであると同時に、隠しファイルであり、未バックアップファイルでもあることを意味しています（**図 2.28** 参照）。

---

* MS-DOS バージョン 3.*x* 以降に付属の BACKUP コマンドなどによってバックアップコピーを行ったときにリセットされ、その後何らかの更新やコピーが行われたときにセットされる。

```
    0000  0001………リードオンリーファイル
    0000  0010………隠しファイル
    0000  0100………システムファイル
+)  0010  0000………未バックアップファイル
    ─────────────────────────────
    0010  0111………実際のシステムファイルの属性(27H)
```

図 2.28　実際のシステムファイルに付けられている属性の値

システムファイルのような特別のファイルに対して、たとえばCOMMAND.COMのような通常のファイルの属性の値は20Hとなっています。読み書き可能な一般のファイルの値は、この20Hです。

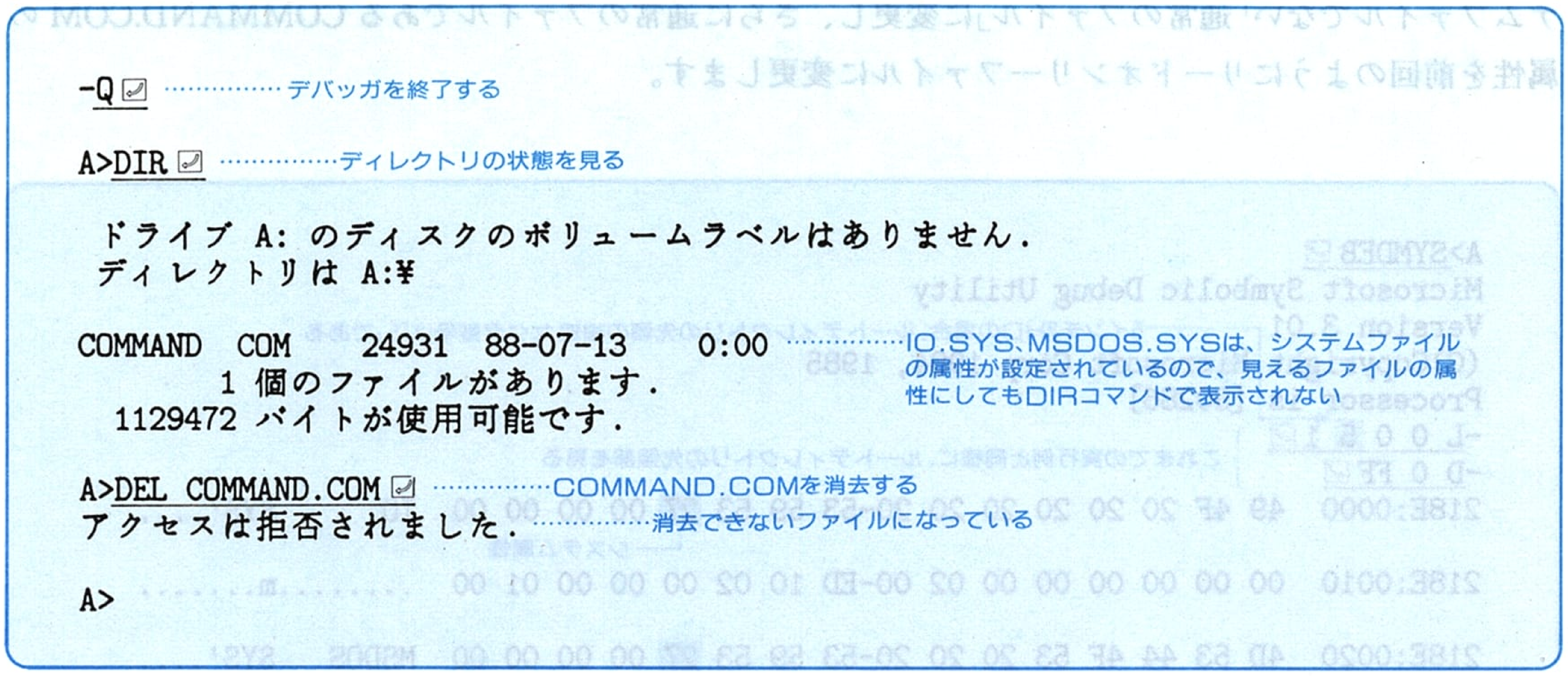
次に、1章で作成したCHMODプログラムを使って、システムファイルであるIO.SYSとMSDOS.SYSの属性を「見えるファイル」に変更することを試み、さらに、通常のファイルであるCOMMAND.COMの属性を「リードオンリーファイル」に変更して、そのあと再度ルートディレクトリの中身をのぞいてみてみましょう。属性の値が変わっていることを確認できるはずです。図2.29にその実行例を示します。

```
A>CHMOD IO.SYS/N↵ ……………IO.SYSを隠しファイルから見えるファイルにする

A>CHMOD MSDOS.SYS/N↵ ……………MSDOS.SYSを隠しファイルから見えるファイルにする

A>CHMOD COMMAND.COM/R↵ ……………通常のファイルであるCOMMAND.COMをリードオンリーファイルにする

A>SYMDEB↵ ……………デバッガを起動する
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 0 5 1↵ }
-D 0 FF↵      } さきの実行例と同様にルートディレクトリの先頭部を見る
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 25 00 00 00 00  IO      SYS%....
                                             27→25
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......

218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 25 00 00 00 00  MSDOS   SYS%....
                                             27→25
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..

218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 21 00 00 00 00  COMMAND COM!....
                                             20→21
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00          m  ca
```

} CHMODプログラムにより、ファイル属性のバイトのビット操作が行われて、それぞれこのように変化している

変更された内容を確認する ⬇

— 図 2.29 —（次ページに続く）



```
-Q↵ ……………デバッガを終了する

A>DIR↵ ……………ディレクトリの状態を見る

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM    24931  88-07-13    0:00 ……………IO.SYS、MSDOS.SYSは、システムファイル
        1 個のファイルがあります.                の属性が設定されているので、見えるファイルの属
  1129472 バイトが使用可能です.                  性にしてもDIRコマンドで表示されない

A>DEL COMMAND.COM↵ ……………COMMAND.COMを消去する
アクセスは拒否されました. ……………消去できないファイルになっている

A>
```

図 2.29　ファイルの属性を CHMOD プログラムで変更し、そのデータの変化を見る

2.3.8　ファイル属性を直接書き換える

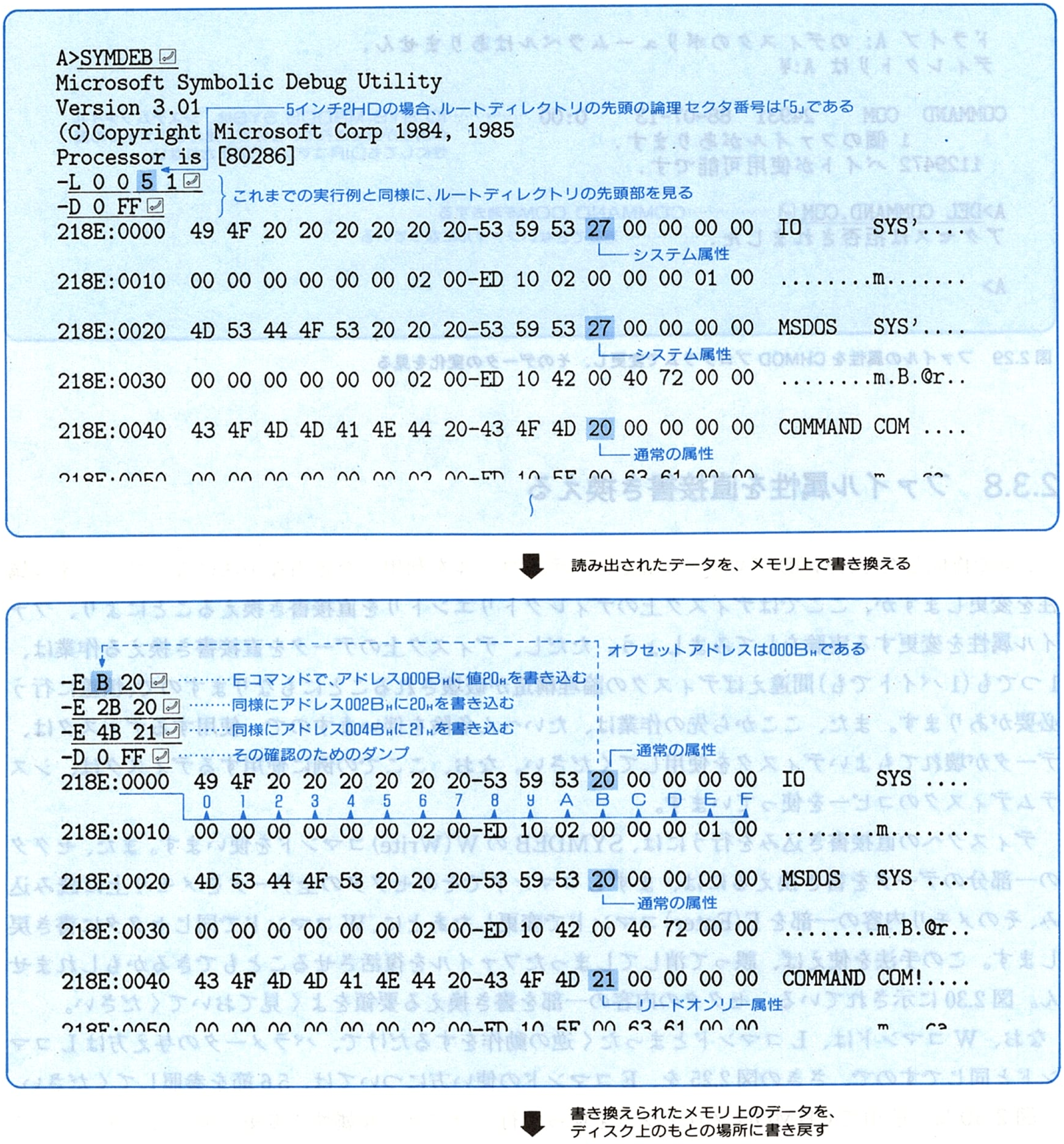
1章で作成したCHMODプログラムは、システムコールを利用した正当な手法によってファイル属性を変更しますが、ここではディスク上のディレクトリエントリを直接書き換えることにより、ファイル属性を変更する実験をしてみましょう。ただし、ディスク上のデータを直接書き換える作業は、1つでも(1バイトでも)間違えばディスクの論理構造が破壊されることにもなりますので、慎重に行う必要があります。また、ここから先の作業は、たいへん危険を伴いますので、使用するディスクは、データが壊れてもよいディスクを使用してください。なお、ここでの例に使用するディスクは、システムディスクのコピーを使っています。

ディスクへの直接書き込みを行うには、SYMDEBのW(Write)コマンドを使います。また、セクタの一部分のデータを書き換えるには、まずLコマンドでそのセクタの全データをメモリ上に読み込み、そのメモリ内容の一部をE(Enter)コマンドで変更したあとに、Wコマンドで同じセクタに書き戻します。この手法を使えば、誤って消してしまったファイルを復活させることもできるかもしれません。図2.30に示されている、セクタの内容の一部を書き換える要領をよく見ておいてください。

なお、Wコマンドは、Lコマンドとまったく逆の動作をするだけで、パラメータの与え方はLコマンドと同じですので、さきの図2.25を、Eコマンドの使い方については、5.6節を参照してください。

図2.30は、前項でCHMODプログラムを使って行ったファイル属性の変更と同じことを、ディスク上のディレクトリを直接書き換えることで実現しようとするものです。今回は、CHMODプログラムではサポートされていない、システムファイルであるIO.SYSおよびMSDOS.SYSの属性を、シス

テムファイルでない「通常のファイル」に変更し、さらに通常のファイルであるCOMMAND.COMの属性を前回のようにリードオンリーファイルに変更します。



```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....

218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......

218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....

218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..

218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....

218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00          m   ca
```

5インチ2HDの場合、ルートディレクトリの先頭の論理セクタ番号は「5」である
これまでの実行例と同様に、ルートディレクトリの先頭部を見る
システム属性
システム属性
通常の属性
読み出されたデータを、メモリ上で書き換える

```
-E B 20⏎
-E 2B 20⏎
-E 4B 21⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 20 00 00 00 00  IO      SYS ....

218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......

218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 20 00 00 00 00  MSDOS   SYS ....

218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..

218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 21 00 00 00 00  COMMAND COM!....

218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00          m   ca
```

オフセットアドレスは000B$_H$である
Eコマンドで、アドレス000B$_H$に値20$_H$を書き込む
同様にアドレス002B$_H$に20$_H$を書き込む
同様にアドレス004B$_H$に21$_H$を書き込む
その確認のためのダンプ
通常の属性
0 1 2 3 4 5 6 7 8 9 A B C D E F
通常の属性
リードオンリー属性
書き換えられたメモリ上のデータを、
ディスク上のもとの場所に書き戻す

— 図2.30 —（次ページに続く）

```
 ┌──────────────── Wコマンド
 │  ┌───────────── ディスクに書き込むメモリ上のデータの先頭アドレス
 │  │  ┌────────── アクセスするドライブ番号(「A：」なら0、「B：」なら1、……)
 │  │  │  ┌─────── 書き込む論理セクタ番号
 │  │  │  │  ┌──── 連続した何セクタ分に書き込むかの値
-W 0 0 5 1↵ ……………Lコマンドで読み出したデータの一部を変更し、再び同じセクタに書き戻す
-Q↵ ……………デバッガを終了する

A>^C ……………ここで必ずCtrl-Cを入力する

A>DIR↵ ……………DIRコマンドで確認する

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

IO        SYS    65536  88-07-13    0:00 ｝システムファイルで見えなかった
MSDOS     SYS    29248  88-07-13    0:00 ｝ものが表示されている
COMMAND   COM    24931  88-07-13    0:00
        3 個のファイルがあります.
  1129472 バイトが使用可能です.

A>CHMOD IO.SYS/?↵ ……………CHMODプログラムを使って、ファイルIO.SYSの属性を確認する

Read/Write File ……………システムファイルであったものが通常のファイルに変更されている

A>CHMOD MSDOS.SYS/?↵ ……………ファイルMSDOS.SYSの属性を確認する

Read/Write File ……………同じく通常のファイルに変更されている

A>CHMOD COMMAND.COM/?↵ ……………ファイルCOMMAND.COMの属性を確認する

Read Only File ……………通常のファイルがリードオンリーファイルに変更されている

A>
```

図 2.30　ファイル属性のバイトを直接書き換え、その結果を確認する

ここで注意！ この結果を確認するためにDIRコマンドを実行する前には、必ずCtrl-Cを入力してください。以前に述べたように、ディレクトリやFATのバッファリングが行われているシステムの場合は、DIRコマンドを実行しても、セクタを直接書き換える以前にバッファリングされていた古い内容が、そのまま表示される可能性があります。SYMDEBのWコマンドによるセクタへの直接書き込みは、MS-DOSのファイルシステムによる管理を受けないので、システムはディスクの内容が変わったことを知ることができないのです。Ctrl-Cを入力すると、ディスクリセットが行われ、バッファリングされていた内容はすべて廃棄されて、次のディスクアクセスに際しては、ディスク上のディレクトリやFATの内容が再度読み込まれます。これは、セクタへの直接書き込みの操作を行った場合に共通

していえることですので、今後の実習においても作業の終了後は、Ctrl-Cの入力を忘れないよう注意してください。

以上の作業で、システムファイルであったIO.SYSとMSDOS.SYSが通常のファイルになり、通常のファイルであったCOMMAND.COMがリードオンリーファイルに変更されました。

ここでの実験のように、セクタの内容を直接書き換えることにより、属性だけでなく、ファイル名、ボリューム名なども書き換えることが可能です。しかし、この方法はあくまで非常手段であり、特別の場合以外行うことではありません。

2.3.9　目的のファイルの中身はどのセクタに？

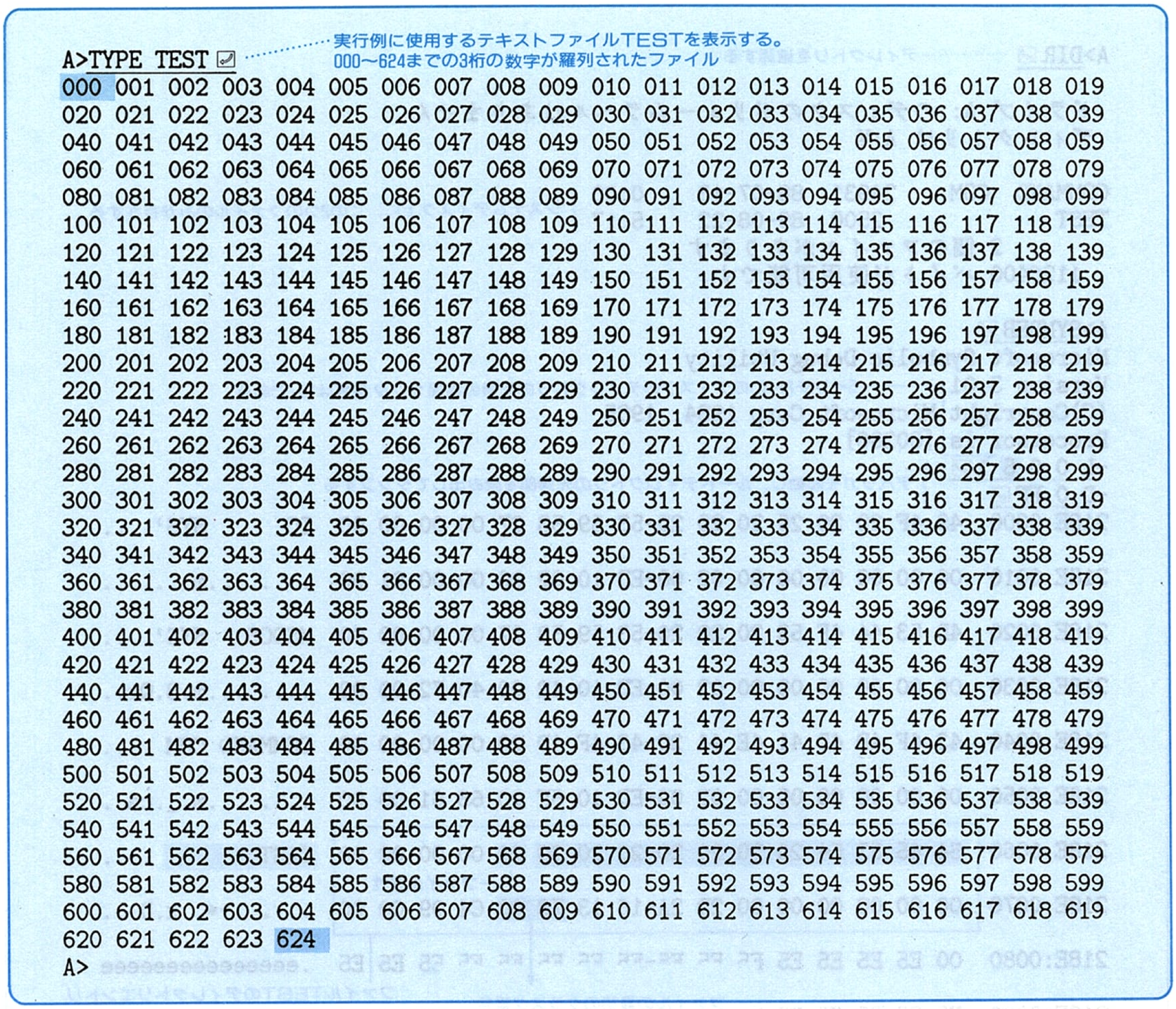
今度は、目的のファイルの中身がディスクのどこに書き込まれているのかを実際にディレクトリやFATをたどって見つけ出してみましょう。また、ファイルを消去するとそのファイルはどうなるのか、ファイルを消去する前後で何が変わるのかを実際に確認してみます。前にも注意しましたが、ここで使うディスクは、破壊されてもよいディスクを用意してください。

まず、消去するためのファイルを作ります。できれば実験のためには、SYMDEBのDコマンドを実行したときに、そのアスキー表示部に文字が表示されるようなテキストファイル(文字ファイル)がよいでしょう。続きのデータのはいっているセクタを、ダンプしながら捜すときにも便利です。ただし、日本語(2バイト系の全角文字)のファイルは、ダンプしても、アスキー表示部には日本語で表示されませんので、ASCIIファイル(1バイト系の半角文字のファイル)でなければ意味がありません。このファイルをわざわざ作るのもたいへんですので、以前作ったプログラムのソースファイルがあればそれを利用すればよいでしょう。いずれにしても、ここでの実験のためのファイルの大きさは3Kバイト程度が適当です。小さすぎて1クラスタで終わっては実験としてつまりませんし、あまり大きいと、クラスタの続きをたどりきれません。

ここでの実行には、この実験のために作成した、クラスタの追跡が楽に行えるような数字だけのテキストファイル「TEST」(サイズは2500バイト)を使います。その内容をTYPEコマンドで示します(図2.31)*。

---

* このファイルは、C言語のプログラムにより作成した。参考までにそのソースファイルをAPPENDIXに示す。



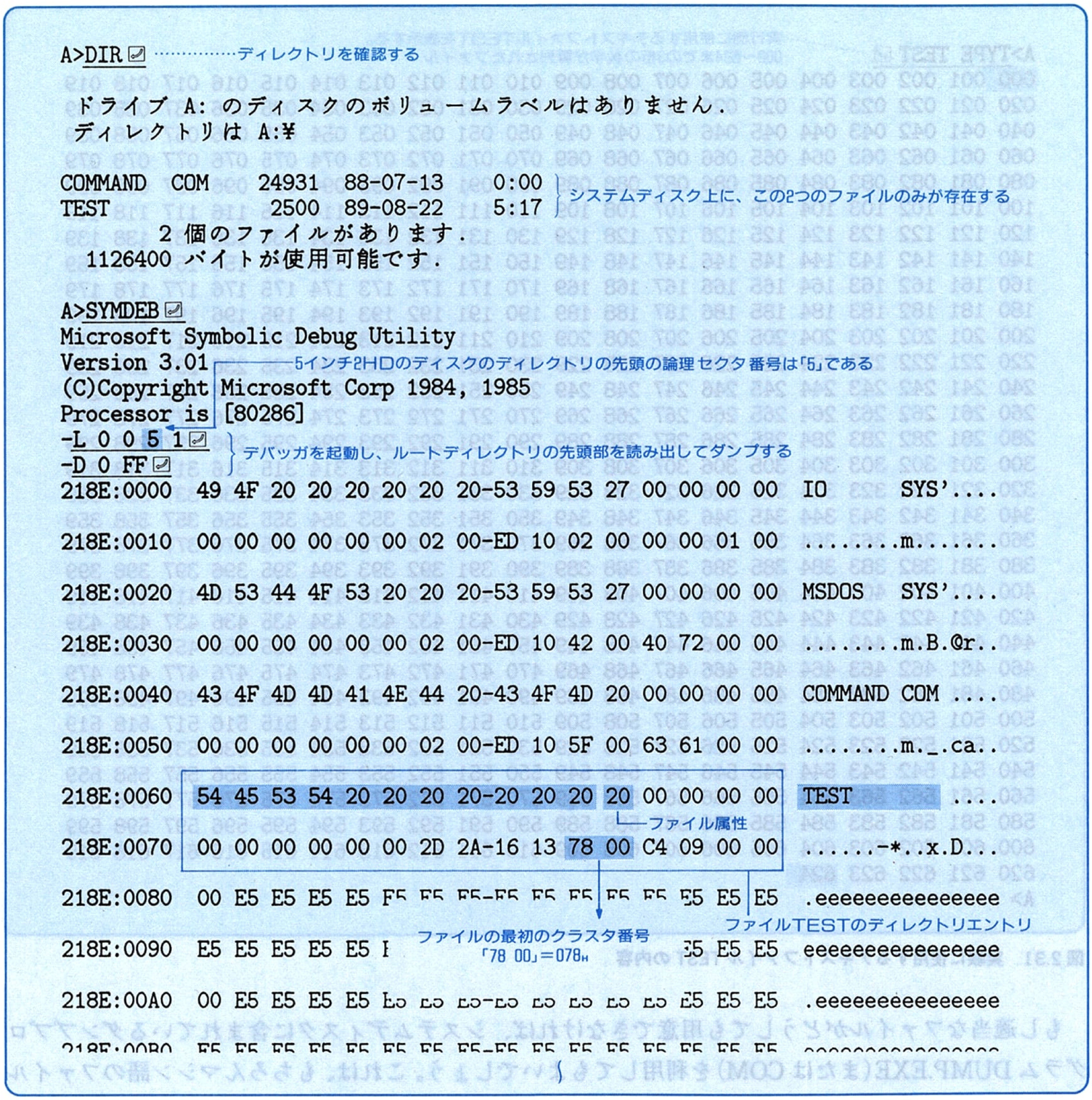
```
A>TYPE TEST ⏎
000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019
020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039
040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059
060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079
080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199
200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219
220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259
260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279
280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299
300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319
320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339
340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379
380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399
400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419
420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439
440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459
460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479
480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499
500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519
520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539
540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559
560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579
580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599
600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619
620 621 622 623 624
A>
```

実行例に使用するテキストファイルTESTを表示する。
000～624までの3桁の数字が羅列されたファイル

図 2.31　実験に使用するテキストファイル TEST の内容

もし適当なファイルがどうしても用意できなければ、システムディスクに含まれているダンププログラム DUMP.EXE（または COM）を利用してもよいでしょう。これは、もちろんマシン語のファイルですが、プログラムの先頭に、この DUMP プログラムのバージョンを表すメッセージがありますので、これを目印にすることができます。ただし、続くクラスタを捜すのは、マシン語ですのでたいへんです。このためにはあらかじめ、DUMP.EXE ファイルを自分自身でダンプしておきましょう。現実にはこんな都合のいいことはありませんが、ここはあくまで実験ですので。

では、この数字だけのテキストファイル TEST の中身が、ディスク上のどこに格納されているかを、ディレクトリや FAT から捜し当てて見てみましょう。まず、DIR コマンドを実行したあと、このディスクのディレクトリをダンプして見てみます（図 2.32）。



```
A>DIR
 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM    24931  88-07-13    0:00
TEST            2500   89-08-22    5:17
        2 個のファイルがあります.
  1126400 バイトが使用可能です.

A>SYMDEB
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1
-D 0 FF
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  54 45 53 54 20 20 20 20-20 20 20 20 00 00 00 00  TEST        ....
218E:0070  00 00 00 00 00 00 2D 2A-16 13 78 00 C4 09 00 00  ......-*..x.D...
218E:0080  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:0090  E5 E5 E5 E5 E5 E                       E5 E5 E5  eeeeeeeeeeeeeeee
218E:00A0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:00B0  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
```

ディレクトリを確認する

システムディスク上に、この2つのファイルのみが存在する

5インチ2HDのディスクのディレクトリの先頭の論理セクタ番号は「5」である

デバッガを起動し、ルートディレクトリの先頭部を読み出してダンプする

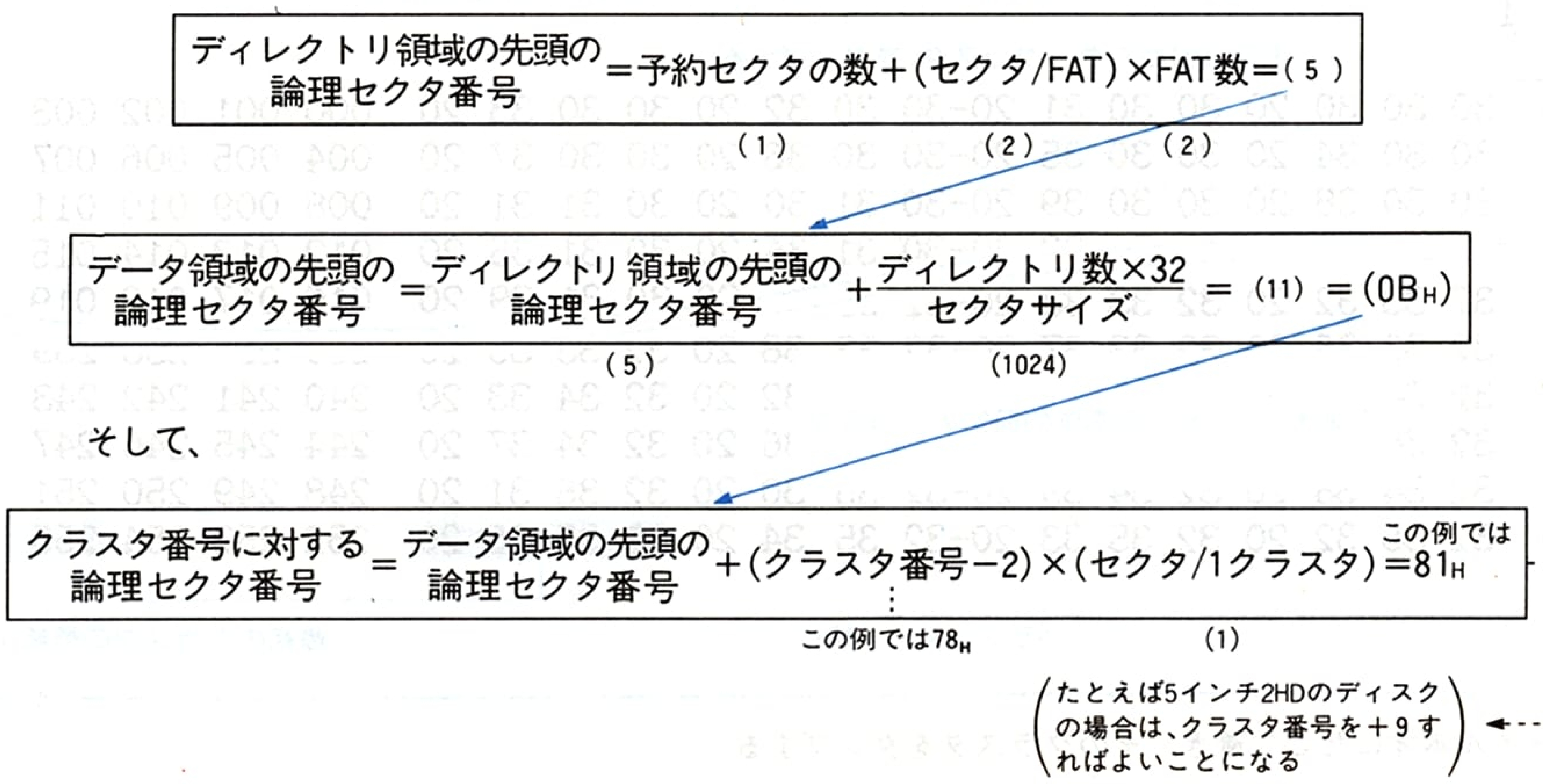
ファイル属性

ファイルの最初のクラスタ番号
「78 00」＝078H

ファイルTESTのディレクトリエントリ

図2.32　実験に使用するディスクのディレクトリをダンプする

このディレクトリのダンプリストを出発点として、ディレクトリやFATによるファイル管理の仕組みを、実際のデータをたどりながら調べていきましょう。まず目的のファイル本体が格納されている場所を知るために、そのファイルの最初のクラスタ番号を調べます(80系CPU(8080、Z80、8086、80286など)のデータ表現形式は、上位バイトと下位バイトの順が逆であることに注意)。クラスタ番号と論理セクタ番号との対応は図2.33のように計算します。


ディレクトリ領域の先頭の論理セクタ番号＝予約セクタの数＋(セクタ/FAT)×FAT数＝(５)
(１)
(２)
(２)
データ領域の先頭の論理セクタ番号＝ディレクトリ領域の先頭の論理セクタ番号＋ディレクトリ数×32／セクタサイズ＝(11)＝(0BH)
(５)
(1024)
そして、
クラスタ番号に対する論理セクタ番号＝データ領域の先頭の論理セクタ番号＋(クラスタ番号−2)×(セクタ/1クラスタ)＝81H
この例では
この例では78H
(1)
(たとえば5インチ2HDのディスクの場合は、クラスタ番号を＋９すればよいことになる)

＊各数値は、表2.1を参照してください。なお、枠外の(　)の値は8インチ2D／5インチ2HD／3.5インチ2HDのものです。

図 2.33　データ領域の先頭の論理セクタ番号と、クラスタ番号に対応する論理セクタ番号を計算する方法

また、１クラスタのサイズは

クラスタのサイズ＝(セクタ/クラスタ)×セクタサイズ

となりますので、ダンプするときにはこのサイズに注意してください。SYMDEB の L コマンドでディスクからロードするときのセクタ数には、「セクタ/クラスタ」を指定するとよいでしょう。

皆さんは、**図 2.34** のように目的のファイル本体の最初のクラスタにたどり着くことができたでしょうか。もしそこに違うデータがはいっていれば、どこかに間違いがあったのでしょう。よく検討してもう一度挑戦してみてください。

ファイルTESTの中身の最初の論理セクタ番号
5インチ2HDのディスクの場合　$0B_H+(78_H-2)\times1=81_H$（←$78_H$を+9する）

0〜$3FF_H$の1セクタ（1クラスタ）をダンプする

```
-L 0 0 81 1⏎
-D 0 3FF⏎
218E:0000  30 30 30 20 30 30 31 20-30 30 32 20 30 30 33 20   000 001 002 003
218E:0010  30 30 34 20 30 30 35 20-30 30 36 20 30 30 37 20   004 005 006 007
218E:0020  30 30 38 20 30 30 39 20-30 31 30 20 30 31 31 20   008 009 010 011
                                  20-30 31 34 20 30 31 35 20   012 013 014 015
218E:03A0  32 33 32 20 32 33 33 20-32          30 31 39 20   016 017 018 019
218E:03B0  32 33 36 20 32 33 37 20-32 33 38 20 32
218E:03C0  32 34                            32 20 32 34 33 20   240 241 242 243
218E:03D0  32 34                            36 20 32 34 37 20   244 245 246 247
218E:03E0  32 34 38 20 32 34 39 20-32 35 30 20 32 35 31 20   248 249 250 251
218E:03F0  32 35 32 20 32 35 33 20-32 35 34 20 32 35 35 20   252 253 254 255
```

ファイルTESTの本体の最初のクラスタ

次のクラスタへ続く

最初のクラスタの最終データ

図2.34　目的のファイル本体にたどり着き、そのクラスタをダンプする

さて、目的のファイルTEST（2500バイト）は、1クラスタのサイズ（5インチ2HDのディスクの場合は1024バイト。これは1セクタでもある）より大きいので、次に続くクラスタが存在するはずです。では、続きのクラスタを捜し出すにはどうしたらよいのでしょうか。これには、前節で解説したようにFATを調べます。FATには続くクラスタの番号が書き込まれています。

FATの読み方はちょっと複雑です。まず、2.3.5で行ったように、もう一度FATの先頭からダンプしてください（図2.35）。「セクタ/FAT」が1より大きいときは、FATの全部をまとめて一度にロードした方がよいでしょう。FATのサイズは「（セクタ/FAT）×セクタサイズ」です。

FATの先頭の論理セクタ番号

FAT

```
-L 0 0 1 2⏎
-D 0 FF⏎
218E:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B   ~...@..`..... ..
218E:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60   @..`..... ..@..`
218E:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02   ..... ..@..`....
218E:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B   ! .#@.%`.'..) .+
218E:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60   @.-`./..1 .3@.5`
218E:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04   .7..9 .;@.=`.?..
218E:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B   Ap.C@.E`.G..I .K
218E:0070  C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 60   @.M`.O..Q .S@.U`
218E:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 FF 0F 06   .W..Y .[@.]`....
218E:0090  61 20 06 63 40 06 65 60-06 67 80 06 69 A0 06 6B   a .c@.e`.g..i .k
218E:00A0  C0 06 6D E0 06 6F 00 07-71 20 07 73 40 07 75 60   @.m`.o..q .s@.u`
218E:00B0  07 77 F0 FF 79 A0 07 FF-0F 00 00 00 00 00 00 00   .wp.y .........
218E:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

図2.35　FATをダンプする

1クラスタに対応するFATエントリは、12ビットから成っています(MS-DOSバージョン3.*x*では、大容量のディスクドライブに対応するため、16ビットを使用することができる)。つまり、3バイトで2クラスタ分のエントリになります。このためFATを読むにはちょっとした操作が必要です。

FATの先頭の1バイトは、FAT IDであり、次の2バイトは現在使用していないダミーで、そこには必ず$\mathrm{FF_H}$がはいっています。つまり、最初の3バイト(これがもし有効であればクラスタ0とクラスタ1の2クラスタ分のエントリになる)はFATのエントリとしての意味を持ちません。つまりクラスタ番号に0と1は存在せず、必ず2から始まります。データ領域の最初のクラスタ番号は2です。さきほどの論理セクタ番号を求める計算式で、クラスタ番号から2を引いているのはこのためなのです。

2クラスタ分のエントリを表す3バイトのFATエントリは、そのデータを**図2.36**のように組み合わせることによって、各クラスタのエントリを表現しています。

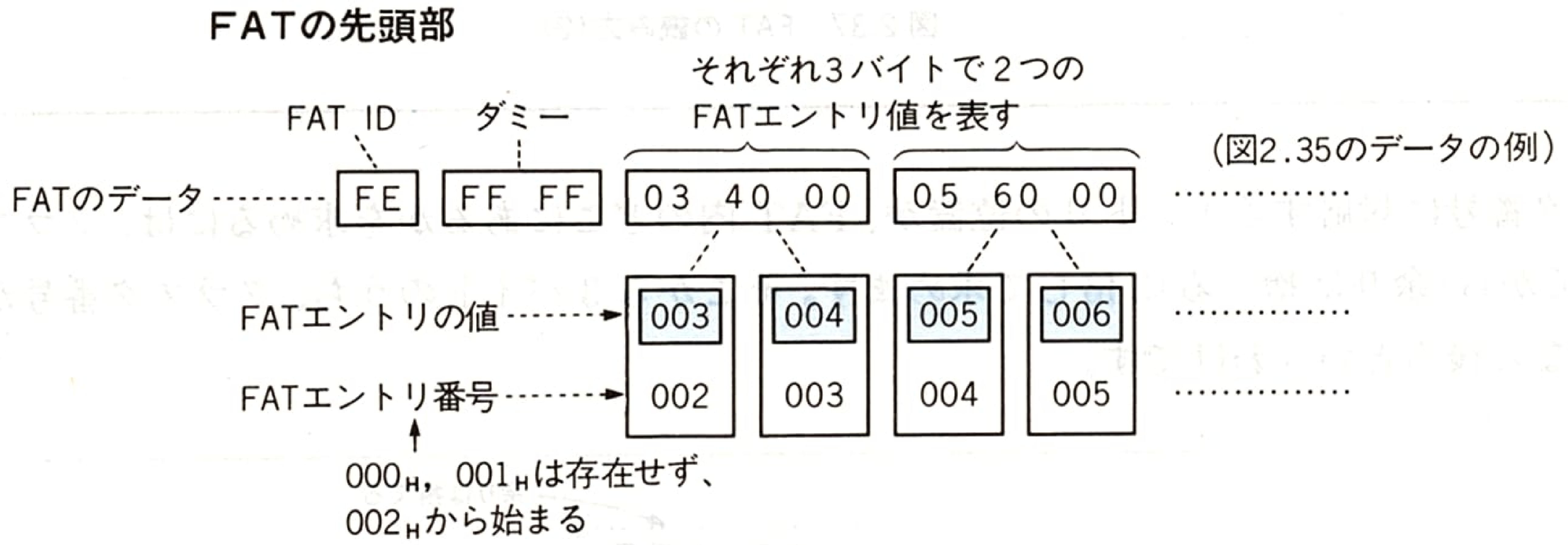

FATの先頭部
それぞれ3バイトで2つの
FATエントリ値を表す
FAT ID
ダミー
(図2.35のデータの例)
FATのデータ
FE
FF FF
03 40 00
05 60 00
FATエントリの値
003
004
005
006
FATエントリ番号
002
003
004
005
000H, 001Hは存在せず、
002Hから始まる

図2.36　FATの読み方(1)

これは、次のように並べ換えて考えるとわかりやすいでしょう。

まず、3バイトのデータを逆順に並べ、そのあとで中央のバイトの上位と下位の4ビットを隣り合う前後のバイトにくっつけて、それぞれ12ビットのデータを2つ作ります。そして、この2組の12ビットのデータを再び逆に並べれば、求めるクラスタの順番どおりのエントリとなるのです。前が偶数番目、後ろが奇数番目のクラスタのエントリとなります(**図2.37**)。

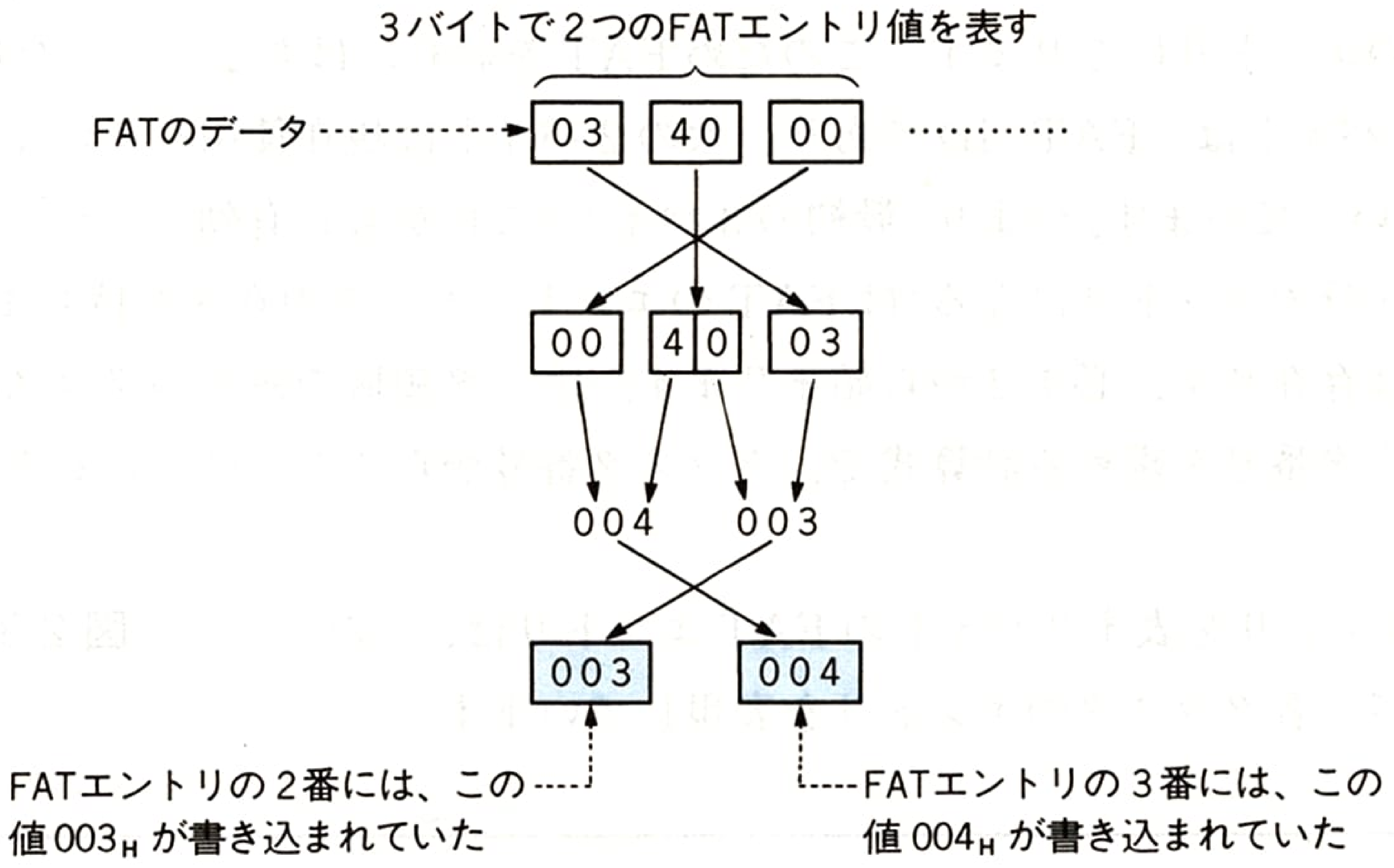
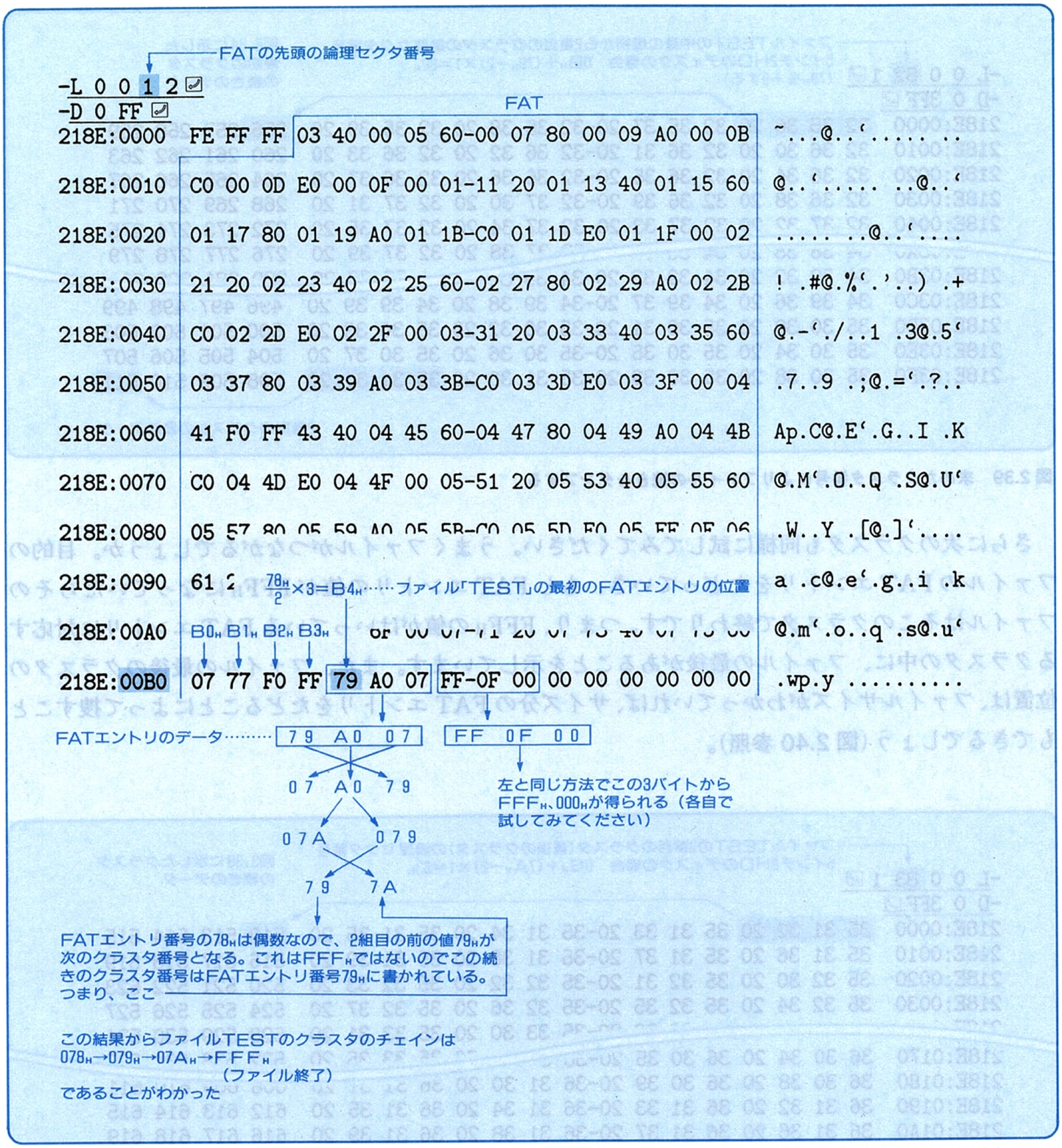

3バイトで2つのFATエントリ値を表す
FATのデータ
03
40
00
……
00
4
0
03
004
003
003
004
FATエントリの2番には、この値003_Hが書き込まれていた
FATエントリの3番には、この値004_Hが書き込まれていた

図 2.37　FATの読み方(2)

クラスタ番号に対応するエントリの位置が、FAT内のどこにあるかを求めるには、クラスタ番号を2で割ってから(余りは捨てる)3倍して求めます。そこから3バイトのうち、クラスタ番号が偶数なら前、奇数なら後ろというわけです。

$$\text{FATエントリの位置} = \overset{\text{余りは捨てる}}{\boxed{\dfrac{\text{クラスタ番号}}{2}}} \times 3$$

このことを応用して、ディレクトリエントリ内の最初のファイルのクラスタ番号に対応するFATエントリの値を求めてみましょう(**図 2.38**)。

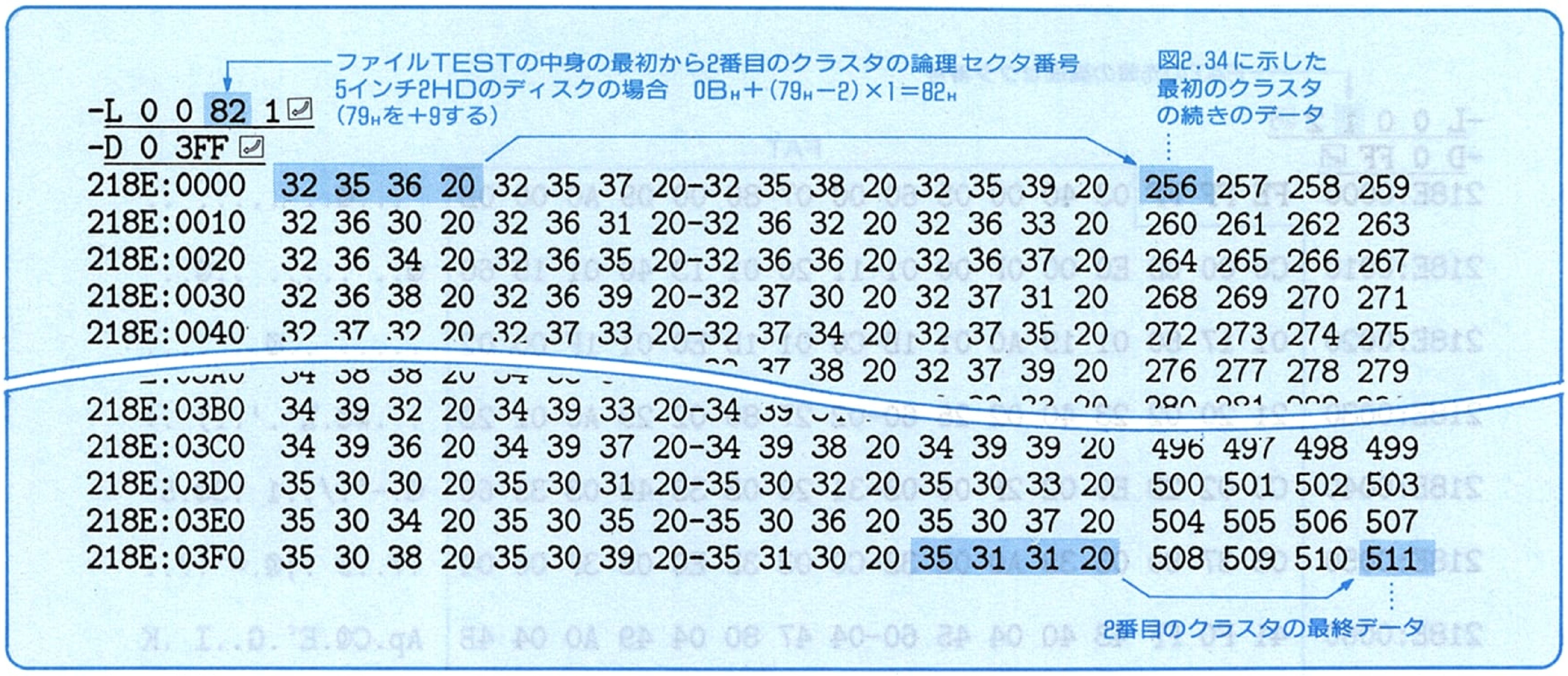


```
-L 0 0 1 2
-D 0 FF
218E:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B  ~...@..‘.......
218E:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60  @..‘..... ..@..‘
218E:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02  ..... ..@..‘....
218E:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B  ! .#@.%‘.’..) .+
218E:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60  @.-‘./..1 .3@.5‘
218E:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04  .7..9 .;@.=‘.?..
218E:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B  Ap.C@.E‘.G..I .K
218E:0070  C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 60  @.M‘.O..Q .S@.U‘
218E:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 FF 0F 06  .W..Y .[@.]‘....
218E:0090  61 2                                              a .c@.e‘.g..i .k
218E:00A0              6F 00 07-71 20 07 73 40 07 75 60  @.m‘.o..q .s@.u‘
218E:00B0  07 77 F0 FF 79 A0 07 FF-0F 00 00 00 00 00 00 00  .wp.y .........
```

FATの先頭の論理セクタ番号
FAT
$\frac{78_H}{2}$×3=B4ₕ……ファイル「TEST」の最初のFATエントリの位置
B0ₕ B1ₕ B2ₕ B3ₕ
FATエントリのデータ………
79 A0 07
FF 0F 00
07 A0 79
07A 079
79 7A
左と同じ方法でこの3バイトから FFFₕ、000ₕが得られる（各自で試してみてください）
FATエントリ番号の78ₕは偶数なので、2組目の前の値79ₕが次のクラスタ番号となる。これはFFFₕではないのでこの続きのクラスタ番号はFATエントリ番号79ₕに書かれている。つまり、ここ
この結果からファイルTESTのクラスタのチェインは
078ₕ→079ₕ→07Aₕ→FFFₕ
（ファイル終了）
であることがわかった

図 2.38　実験用ファイルの FAT エントリ値の計算例

この値は、目的のファイルの、現在のクラスタに続くクラスタの番号になります。このクラスタ番号に対応する論理セクタ番号を求めてダンプしてみましょう（図 2.39）。ファイルの続きのデータになっているでしょうか。



```
-L 0 0 82 1⏎
-D 0 3FF⏎
218E:0000  32 35 36 20 32 35 37 20-32 35 38 20 32 35 39 20  256 257 258 259
218E:0010  32 36 30 20 32 36 31 20-32 36 32 20 32 36 33 20  260 261 262 263
218E:0020  32 36 34 20 32 36 35 20-32 36 36 20 32 36 37 20  264 265 266 267
218E:0030  32 36 38 20 32 36 39 20-32 37 30 20 32 37 31 20  268 269 270 271
218E:0040  32 37 32 20 32 37 33 20-32 37 34 20 32 37 35 20  272 273 274 275
                                      37 38 20 32 37 39 20  276 277 278 279
218E:03B0  34 39 32 20 34 39 33 20-34
218E:03C0  34 39 36 20 34 39 37 20-34 39 38 20 34 39 39 20  496 497 498 499
218E:03D0  35 30 30 20 35 30 31 20-35 30 32 20 35 30 33 20  500 501 502 503
218E:03E0  35 30 34 20 35 30 35 20-35 30 36 20 35 30 37 20  504 505 506 507
218E:03F0  35 30 38 20 35 30 39 20-35 31 30 20 35 31 31 20  508 509 510 511
```

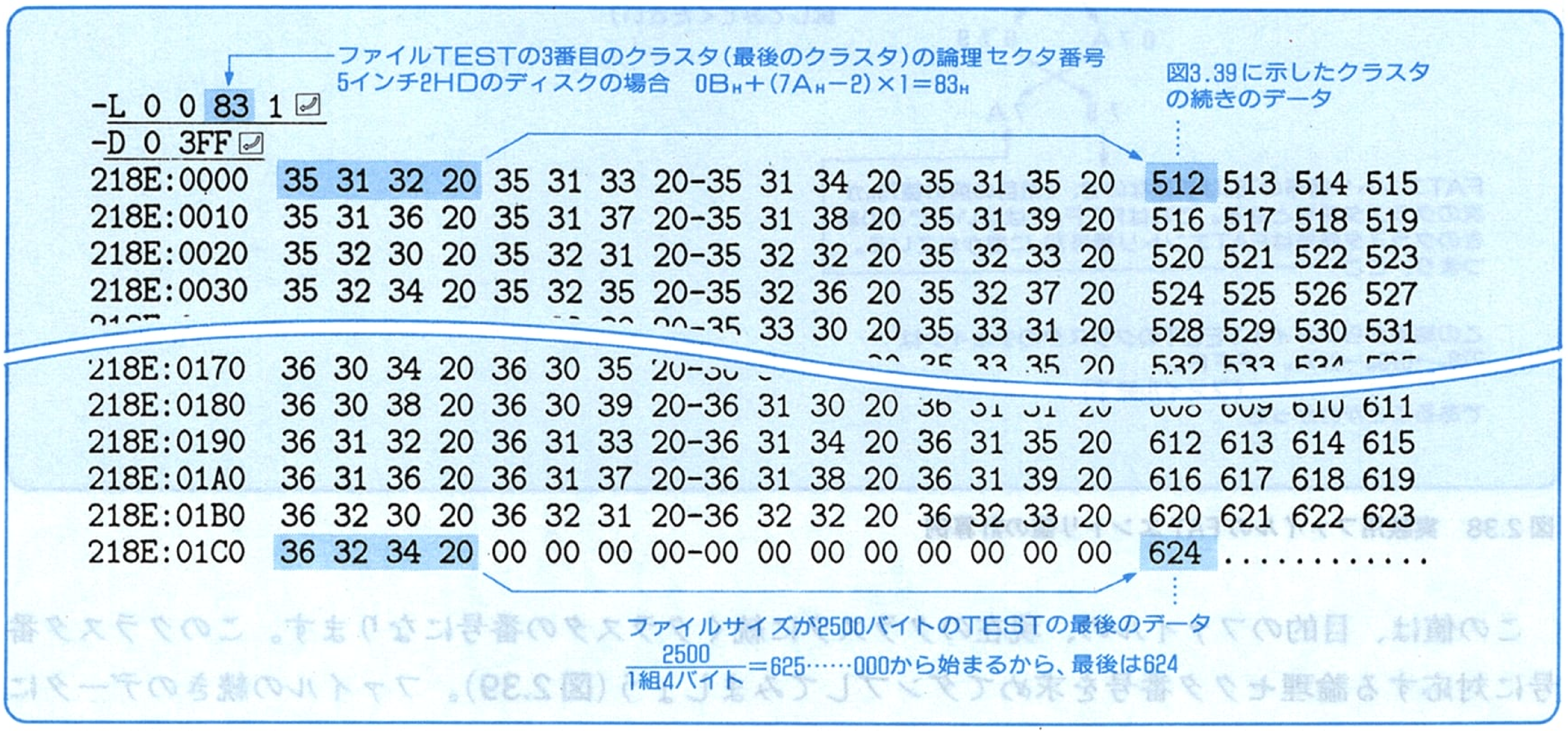
ファイルTESTの中身の最初から2番目のクラスタの論理セクタ番号
5インチ2HDのディスクの場合　$0B_H+(79_H-2)\times1=82_H$
（79_Hを+9する）

図2.34に示した
最初のクラスタ
の続きのデータ

2番目のクラスタの最終データ

図 2.39　求めたクラスタ番号によりファイルの続きをダンプする

さらに次のクラスタも同様に試してみてください。うまくファイルがつながるでしょうか。目的のファイルのFATエントリをたどっていき、もしFATエントリの値がFFF_Hになっていたらそのファイルはそこのクラスタで終わりです。つまり、FFF_Hの値がはいっているFATエントリに対応するクラスタの中に、ファイルの最後があることを示しています。また、ファイルの最後のクラスタの位置は、ファイルサイズがわかっていれば、サイズ分のFATエントリをたどることによって捜すこともできるでしょう（図2.40参照）。



```
-L 0 0 83 1⏎
-D 0 3FF⏎
218E:0000  35 31 32 20 35 31 33 20-35 31 34 20 35 31 35 20  512 513 514 515
218E:0010  35 31 36 20 35 31 37 20-35 31 38 20 35 31 39 20  516 517 518 519
218E:0020  35 32 30 20 35 32 31 20-35 32 32 20 35 32 33 20  520 521 522 523
218E:0030  35 32 34 20 35 32 35 20-35 32 36 20 35 32 37 20  524 525 526 527
                                -35 33 30 20 35 33 31 20  528 529 530 531
218E:0170  36 30 34 20 36 30 35 20-
218E:0180  36 30 38 20 36 30 39 20-36 31 30 20                    610 611
218E:0190  36 31 32 20 36 31 33 20-36 31 34 20 36 31 35 20  612 613 614 615
218E:01A0  36 31 36 20 36 31 37 20-36 31 38 20 36 31 39 20  616 617 618 619
218E:01B0  36 32 30 20 36 32 31 20-36 32 32 20 36 32 33 20  620 621 622 623
218E:01C0  36 32 34 20 00 00 00 00-00 00 00 00 00 00 00 00  624 ............
```

ファイルTESTの3番目のクラスタ（最後のクラスタ）の論理セクタ番号
5インチ2HDのディスクの場合　$0B_H+(7A_H-2)\times1=83_H$

図3.39に示したクラスタ
の続きのデータ

ファイルサイズが2500バイトのTESTの最後のデータ
$\frac{2500}{\text{1組4バイト}}=625$……000から始まるから、最後は624

図 2.40　次のクラスタ番号を求め、ファイルの続きをダンプする

■16ビットFAT　―ハードディスクを有効に活用するための機能―

バージョン2.xx以前のMS-DOSのファイルシステム(いわゆる12ビットFATシステム)は、フロッピーディスクの使用を前提として設計されていました。そのため、現在では一般に使われるようになったハードディスクのような大容量のメディアを効率よく管理するには無理があります。

この限界を破り大容量のハードディスクに対応するために、バージョン3.xのMS-DOSには新しい機能が追加されています。それは、12ビット単位だけではなく、「16ビット単位」のFAT――いわゆる**16ビットFAT**を扱えるようにしたことです。

容量20M(メガ)バイトのハードディスクは、フロッピーディスク20枚分のファイルを格納することができそうですが、実際には空白の領域がたくさんできてしまい、20Mバイトすべてをまるまるファイル領域として使えるわけではありません。これはFATのエントリ数の制限から、大容量ディスクではクラスタサイズが大きくなってしまうからです。

FATの1エントリを12ビットで表すと、FATの1エントリで表せるクラスタ番号は002Hから FF6Hまでの4,085個です。20Mバイトのハードディスクでは、1クラスタのサイズを8Kバイトにしなければこの数に収まりません。

これはどういうことかというと、たった2,3バイトからなる小さなファイルを作ったとしても、ディスクの中では8Kバイトの領域を占有するということです。7Kバイト以上の領域が使われないまま無駄になってしまいます。バッチファイルのような小さなファイルをたくさん作ると、使われない(使えない)領域がどんどん増えていきます(**図2.41**参照)。

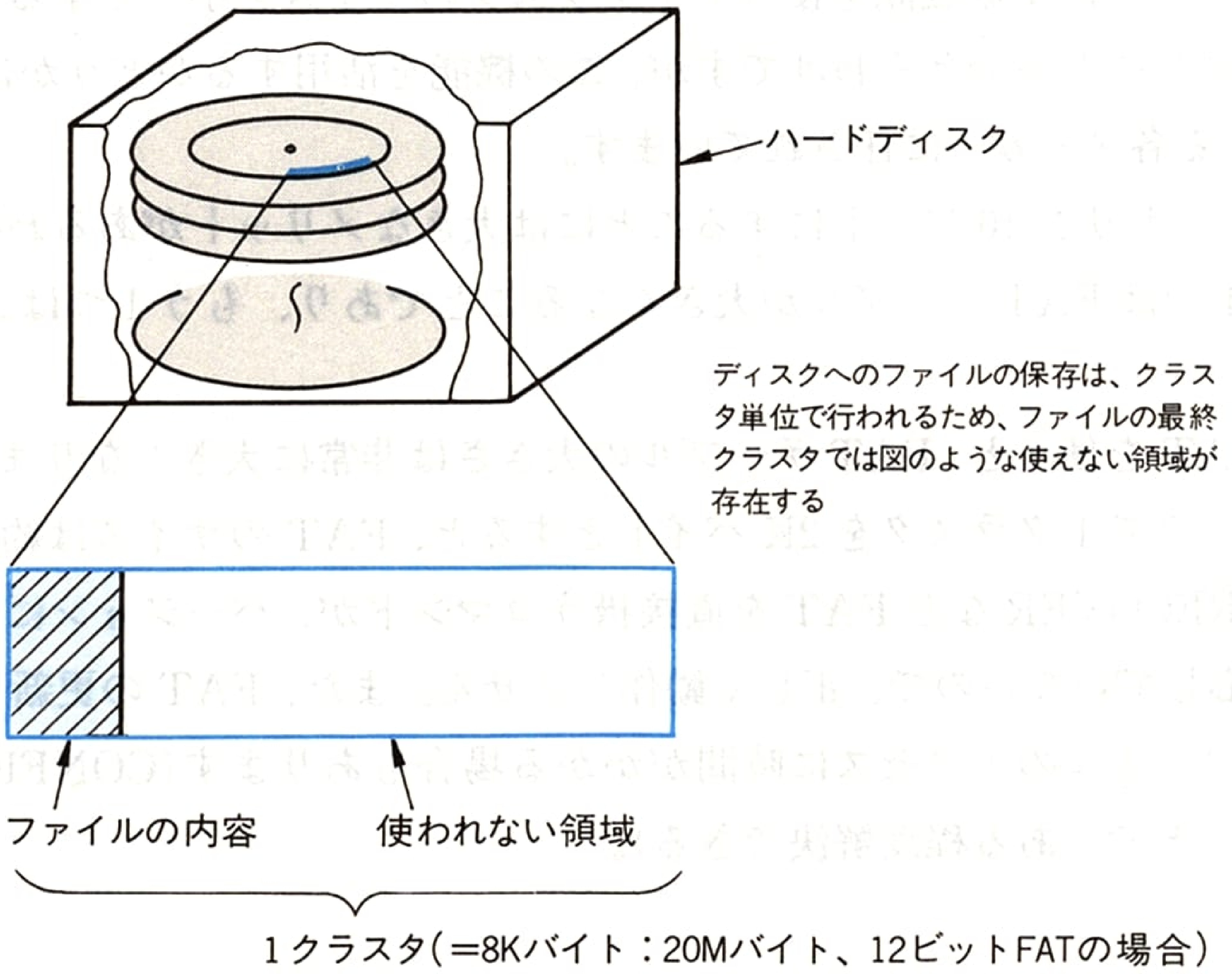

ハードディスク
ディスクへのファイルの保存は、クラスタ単位で行われるため、ファイルの最終クラスタでは図のような使えない領域が存在する
ファイルの内容
使われない領域
1クラスタ(＝8Kバイト：20Mバイト、12ビットFATの場合)

図2.41　空白のディスク領域

このような無駄を解消するために導入されたのが16ビット単位のFATです。FATの1エントリを12ビットから16ビットに拡張することにより、FATの1エントリで表せるクラスタ番号は0002_{H}～$FFF6_{H}$の65,525個となります。これだけあれば1クラスタのサイズを2Kバイトや1Kバイトにしても、十分カバーできますし、40Mバイト以上の大容量ハードディスクにも容易に対応できます。FATの1エントリを16ビットで表す方法のことを**16ビットFAT**と呼んでいます(図**2.42**参照)。

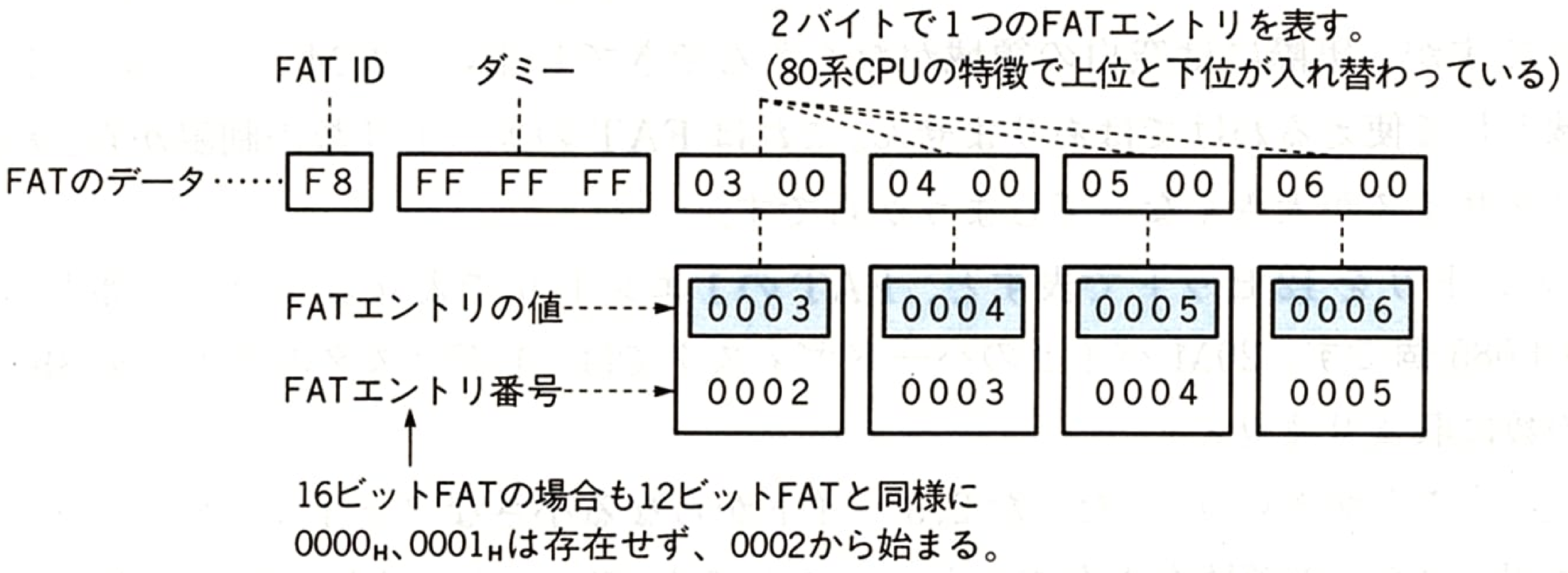

2バイトで1つのFATエントリを表す。
(80系CPUの特徴で上位と下位が入れ替わっている)
FAT ID
ダミー
FATのデータ
F8
FF FF FF
03 00
04 00
05 00
06 00
FATエントリの値
0003
0004
0005
0006
FATエントリ番号
0002
0003
0004
0005
16ビットFATの場合も12ビットFATと同様に
0000_{H}、0001_{H}は存在せず、0002から始まる。

図2.42　16ビットFAT

FATを16ビットにする機能を使って1クラスタのサイズを小さくすることができ、ディスク容量を有効に活用することができるわけですが、この機能を活用するかどうかはIO.SYS(詳しくは3章で解説)を作成する各メーカーに任されています。

FATの1エントリを16ビットにすることには大きなメリットがあるわけですが、いくつかの問題があります。1つはFATテーブルが大きくなることであり、もう1つはシステムの互換性の問題です。

16ビットFATを使うと、FATテーブルの大きさは非常に大きくなります。たとえば、40Mバイトのハードディスクで1クラスタを2Kバイトとすると、FATのサイズは約40Kバイトにもなります。CHKDSKやRECOVERなどFATを直接扱うコマンドが、バージョンによっては、このように大きなFATに対応していないので、正しく動作しません。また、FATの更新などの処理に時間がかかるようになり、ファイルのアクセスに時間がかかる場合もあります(CONFIG.SYSのBUFFERS指定を大きくすることで、ある程度解決できる)。

また、従来の12ビットFATしか対応していないシステムのフロッピーディスクなどで立ち上げて、16ビットFATにフォーマット処理したハードディスクをアクセスすると、正しくアクセスでき

ずに場合によっては破壊してしまいます。このような互換性の問題から、従来機種のハードディスクでは16ビットFATの導入を見送り、古いバージョンのMS-DOSではサポートされていない新しい機種のハードディスクからのみ16ビットFATをサポートしているメーカーもあります*。

2.3.10 消去したファイルの行方は？

さてみなさんは、重要なファイルをついうっかり消去してしまい、泣くに泣けない思いをされたことはありませんか？ このような場合、なんとか消去したファイルを生き返らせる方法はないものか、と思うでしょう。

実は、ファイルの消去という処理は、ファイル本体のデータを消去してしまうわけではなくディレクトリやFATからそのファイルに関する情報を消去してしまうだけなので、ファイルの中身だけはそのまま残っています(図**2.43**)。ですから、忍耐と努力を厭わなければ、一度消去したファイルを復活することも不可能ではありません。ただし、それが可能なのはファイルを消去した直後の話であり、消したあとに何らかの書き込みを行ってしまえばファイル復活の望みはほとんどないでしょう。

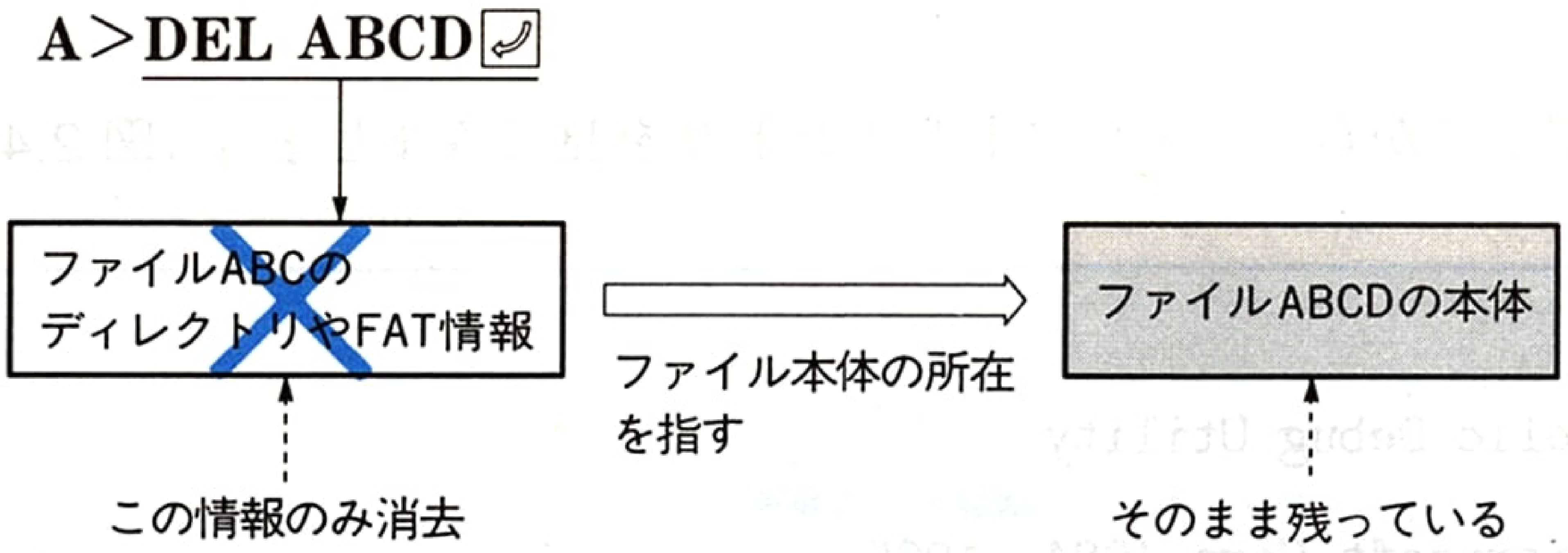
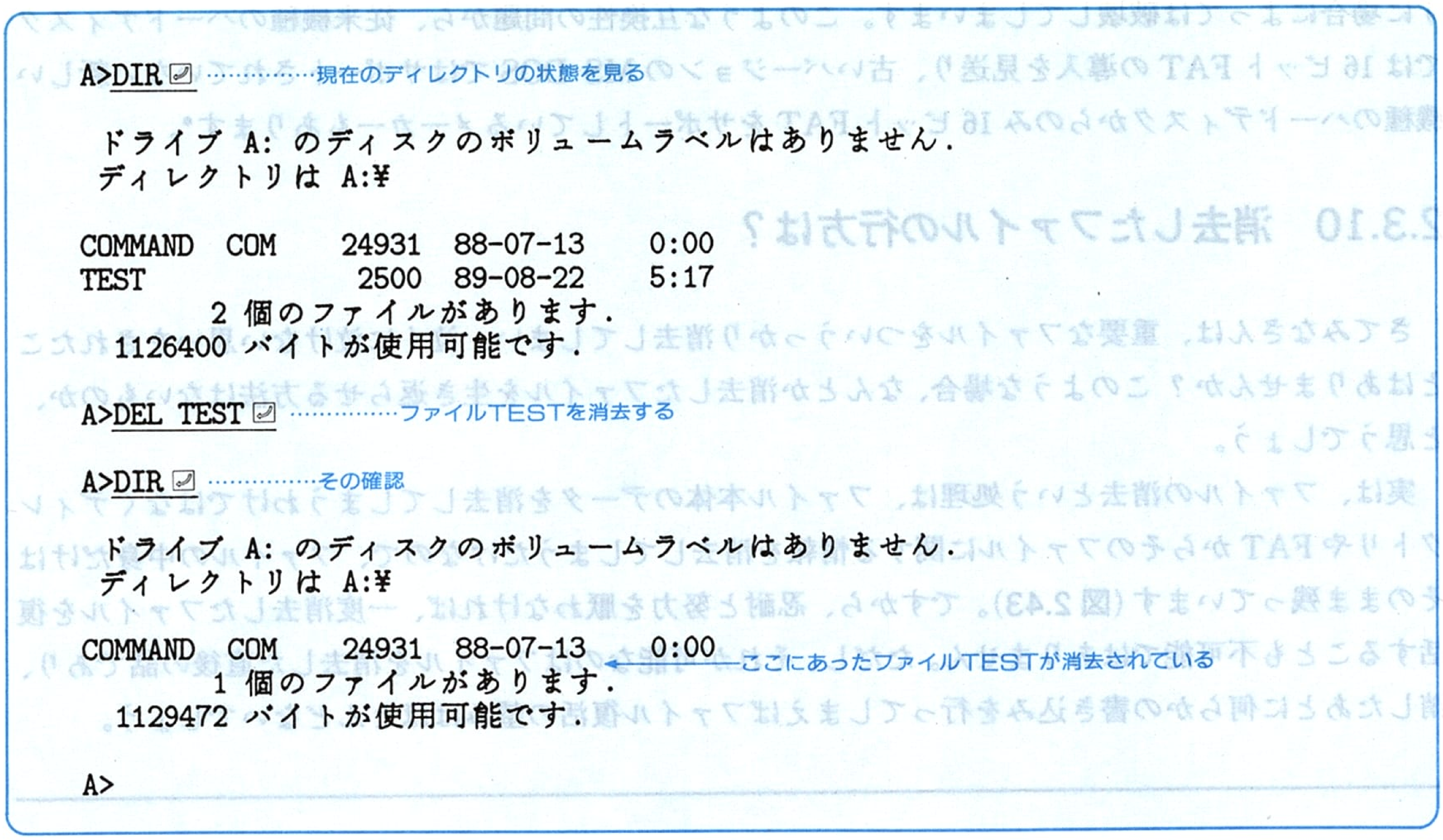


図2.43 ファイルを消去してもファイルの本体は残っている

では、ファイルを消去した場合、ディスク上にどのような変化が起こるのかを調べてみましょう。まず、今までの実験に使っていたファイルTESTを消去します(図**2.44**)。

---

* このようなメーカーのMS-DOSであっても、バージョンが3.*x*以降ならばMS-DOS自体は16ビットFATに対応しているので、IO.SYS中のハードディスクのデバイスドライバを自分で書くか、パッチを当てるなどすれば、16ビットFATに改造することができる。しかし、12ビットFAT-16ビットFAT間のトラブルは意識して避ける以外にはない。



```
A>DIR⏎

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13    0:00
TEST              2500  89-08-22    5:17
        2 個のファイルがあります.
  1126400 バイトが使用可能です.

A>DEL TEST⏎

A>DIR⏎

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13    0:00
        1 個のファイルがあります.
  1129472 バイトが使用可能です.

A>
```

現在のディレクトリの状態を見る

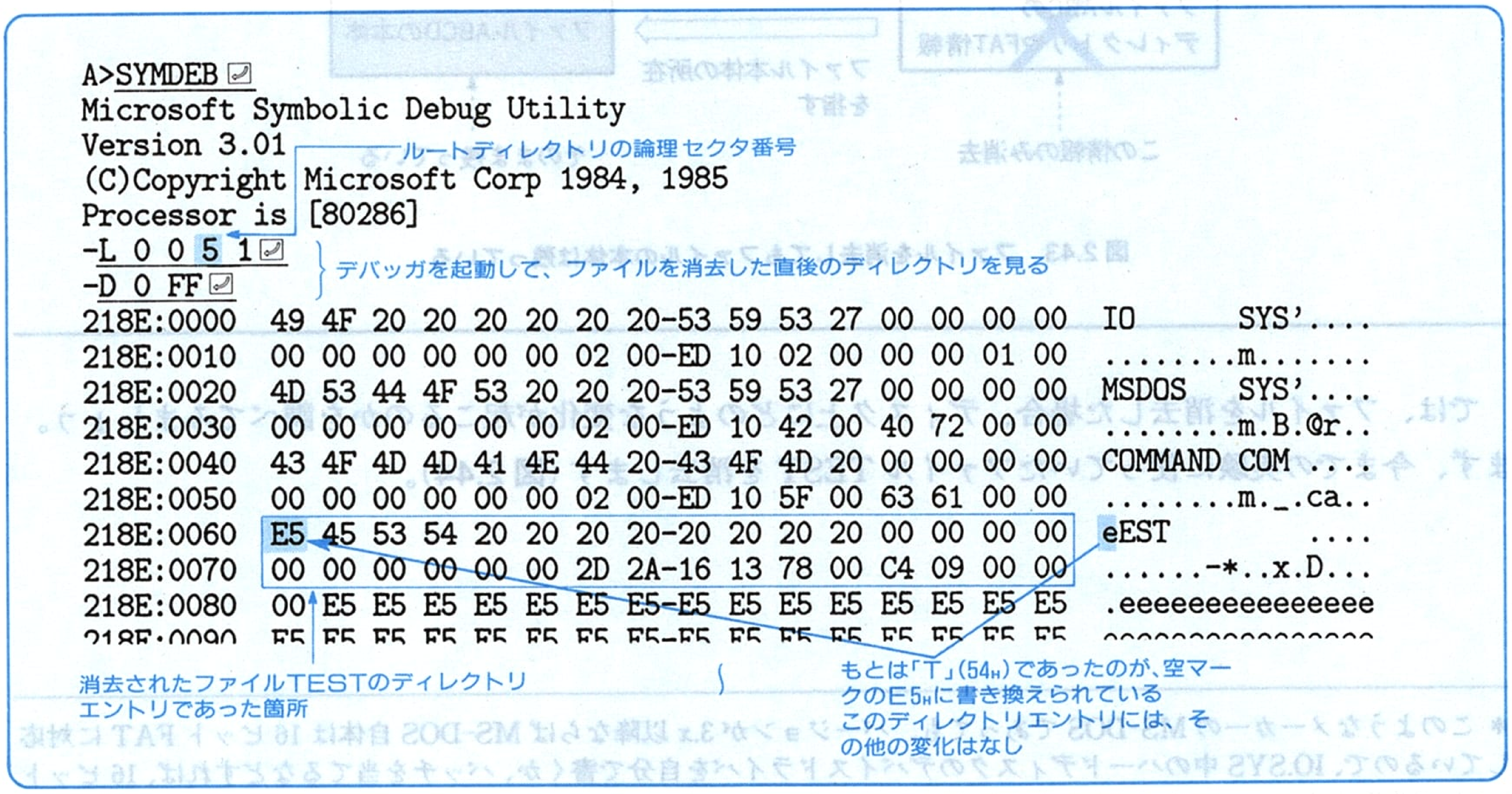
ファイルTESTを消去する

その確認

ここにあったファイルTESTが消去されている

図2.44　実験用のファイルTESTを消去する

消去されたのを確認してから、ディレクトリエントリを見てみましょう（図2.45）。



```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  E5 45 53 54 20 20 20 20-20 20 20 20 00 00 00 00  eEST        ....
218E:0070  00 00 00 00 00 00 2D 2A-16 13 78 00 C4 09 00 00  ......-*..x.D...
218E:0080  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:0090  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
```

ルートディレクトリの論理セクタ番号

デバッガを起動して、ファイルを消去した直後のディレクトリを見る

消去されたファイルTESTのディレクトリエントリであった箇所

もとは「T」(54_H)であったのが、空マークのE5_Hに書き換えられている
このディレクトリエントリには、その他の変化はなし

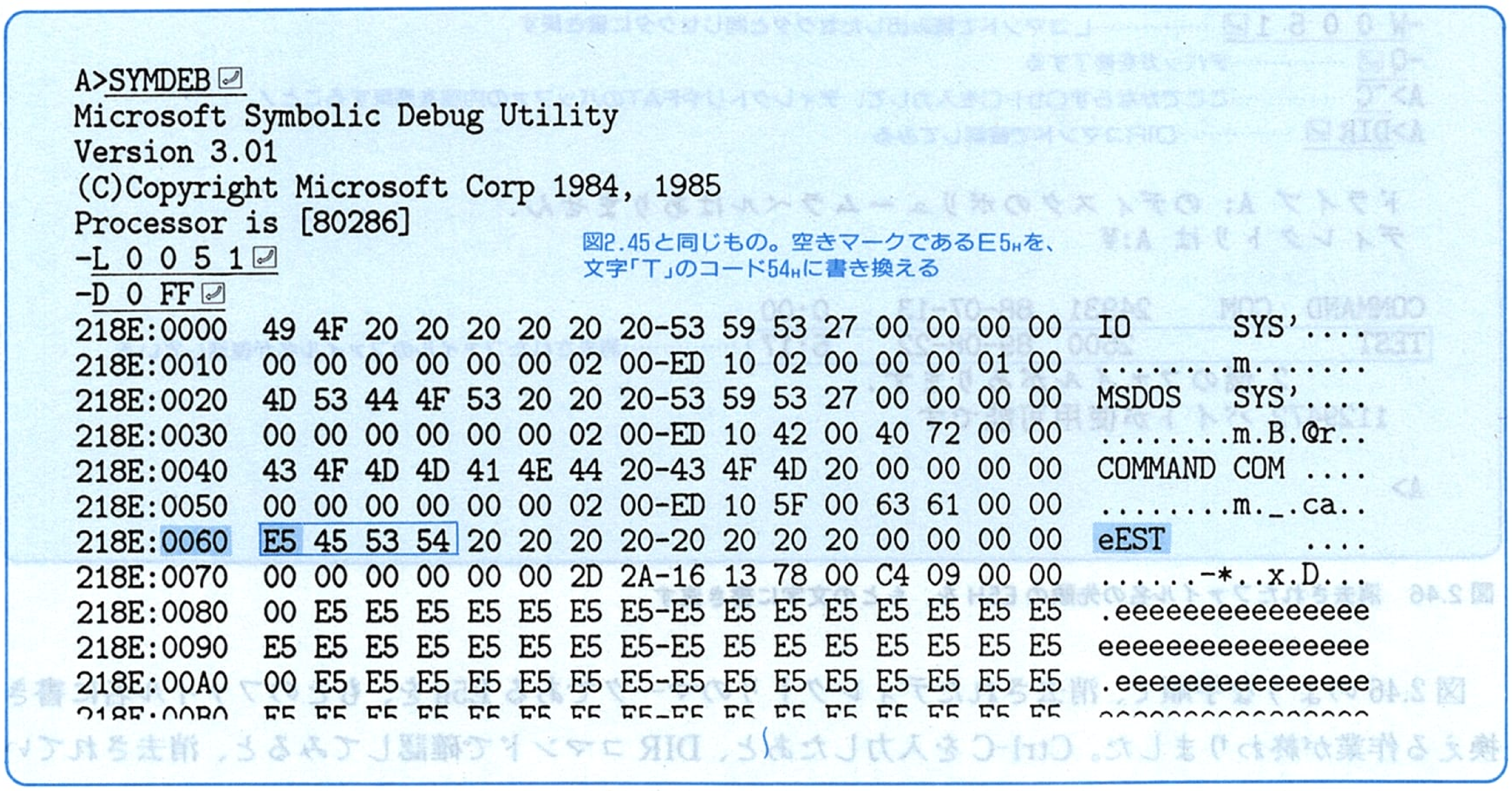
図2.45　ファイルTESTを消去したあとのディレクトリエントリを見る

このように、消去したファイルのディレクトリエントリのファイル名の先頭バイトが、E5Hになっています。ディレクトリエントリは、それ以外何も変わっていないはずです。消去前の状態(図2.32)と比較してみてください。

ディレクトリエントリの先頭がE5Hであれば、そのディレクトリエントリは使用されていない(空いている)ことを示します。ファイルを消去するということは、そのファイルのディレクトリエントリの先頭に、「空きマーク」のE5Hを書き込み、そのエントリを解放するだけのことなのです。ファイルの本体部は、まったく触れられることはなく、そのままの状態で残っています。

また、このディレクトリエントリの先頭が00Hであれば、そこから先には有効なエントリが存在しないことを示します。この00Hを見つけたらそれ以上先はディレクトリの検索をしないようにして、むだなディスクアクセスを防いでいるのです。

ではいよいよ、消去してしまったファイルを復活させる作業に取りかかりましょう。具体的には、E5Hに書き換えられた、そのファイルのディレクトリエントリの先頭バイトを、もとの値(もとのファイル名の頭の文字コード)に戻せばよいのです。これだけの操作で、はたしてファイルが復活するでしょうか。この書き換えには、本章の2.3.4で述べた、セクタデータの一部を書き換える方法を使います(図2.46)。

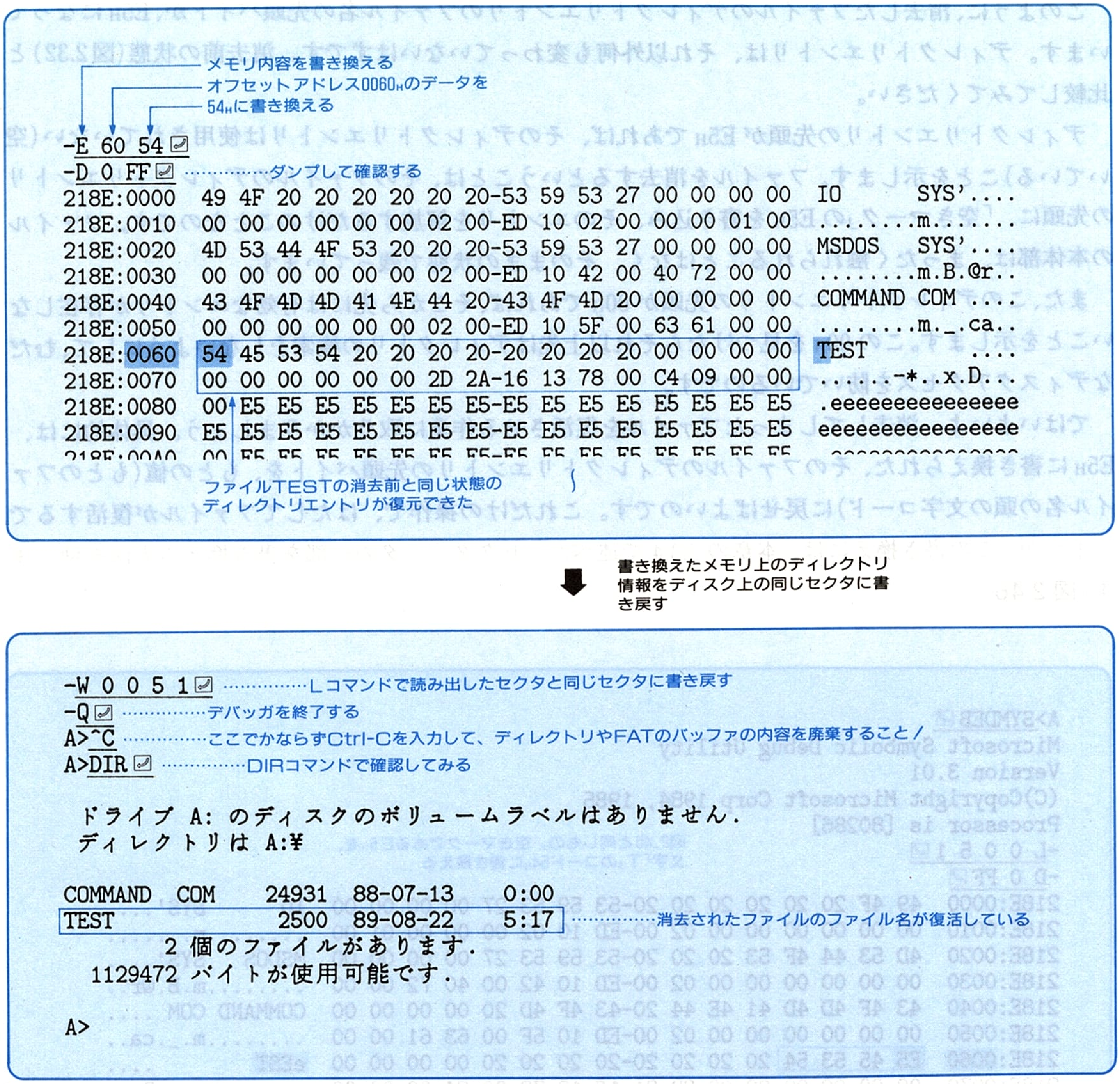


```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  E5 45 53 54 20 20 20 20-20 20 20 20 00 00 00 00  eEST        ....
218E:0070  00 00 00 00 00 00 2D 2A-16 13 78 00 C4 09 00 00  ......-*..x.D...
218E:0080  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:0090  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
218E:00A0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
```

図2.45と同じもの。空きマークであるE5Hを、文字「T」のコード54Hに書き換える

— 図2.46 —（次ページに続く）

メモリ上にロードしたディレクトリの情報を書き換える



```
-E 60 54⏎
-D 0 FF⏎
218E:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
218E:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
218E:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
218E:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
218E:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
218E:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
218E:0060  54 45 53 54 20 20 20 20-20 20 20 20 00 00 00 00  TEST        ....
218E:0070  00 00 00 00 00 00 2D 2A-16 13 78 00 C4 09 00 00  ......-*..x.D...
218E:0080  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  .eeeeeeeeeeeeeee
218E:0090  E5 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5  eeeeeeeeeeeeeeee
218E:00A0  00 E5 E5 E5 E5 E5 E5 E5-E5 E5 E5 E5 E5 E5 E5 E5
```

```
-W 0 0 5 1⏎
-Q⏎
A>^C
A>DIR⏎

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13   0:00
TEST              2500  89-08-22   5:17
        2 個のファイルがあります.
   1129472 バイトが使用可能です.

A>
```

メモリ内容を書き換える
オフセットアドレス0060Hのデータを
54Hに書き換える
ダンプして確認する
ファイルTESTの消去前と同じ状態のディレクトリエントリが復元できた
書き換えたメモリ上のディレクトリ情報をディスク上の同じセクタに書き戻す
Lコマンドで読み出したセクタと同じセクタに書き戻す
デバッガを終了する
ここでかならずCtrl-Cを入力して、ディレクトリやFATのバッファの内容を廃棄すること！
DIRコマンドで確認してみる
消去されたファイルのファイル名が復活している

図2.46　消去されたファイル名の先頭のE5Hを、もとの文字に書き直す

図2.46のような手順で、消去されたディレクトリのマークであるE5Hを、もとのファイル名に書き換える作業が終わりました。Ctrl-Cを入力したあと、DIRコマンドで確認してみると、消去されていたファイルが表示されました。本当にこれでファイルは復活したのでしょうか!?

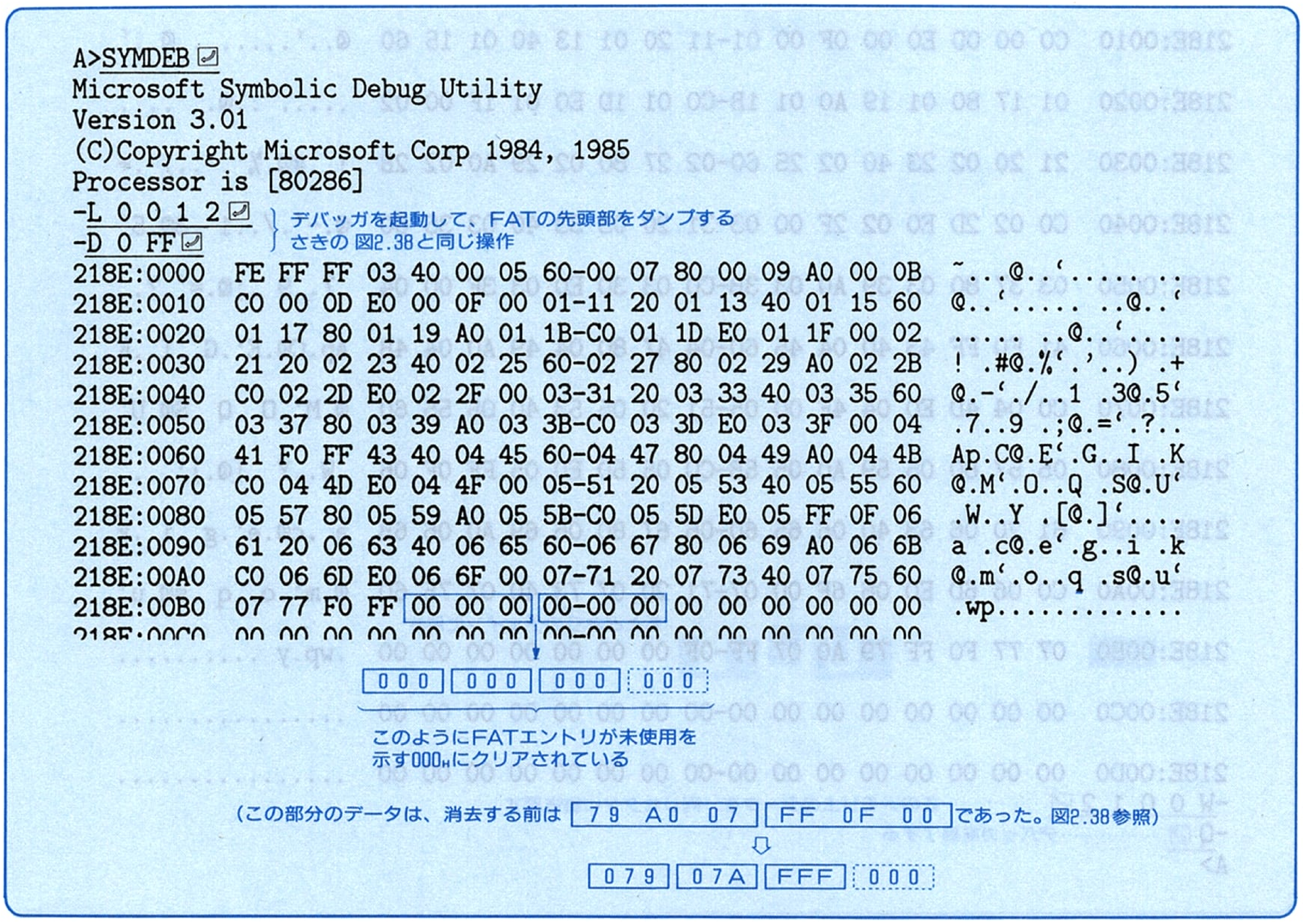
喜ぶのはまだ早すぎます。この復活したはずのファイルTESTの内容を、TYPEコマンドで確認してみましょう（図2.47）。

```
A>TYPE TEST ↵ ……………復活したファイルの中身を表示してみる
0で除算をしました. ……何も表示されないか、あるいはこのようなエラーメッセージなどが表示される
                      つまり、ファイルTESTの中身がシステムに認識されていない
A>
```

図 2.47　復活したはずのファイル内容を表示する

おや？ 何も表示されませんね。ファイル名だけは復活しましたが、このファイルには中身がありません。このような E5H の操作だけで、ファイルが復活するように書いてある参考書もあるようですが、そうはいきません。CP／M ではこれに近い操作だけでファイルが復活するのですが、MS-DOS ではここからがたいへんなのです。

みなさんもすでにお気付きのことと思いますが、FAT のことを忘れていました。では、ファイルを消去することによって、FAT はいったいどのような変化を受けるのでしょうか。まずそれを確認しておきましょう（図 2.48）。



```
A>SYMDEB ↵
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 1 2 ↵  } デバッガを起動して、FATの先頭部をダンプする
-D 0 FF ↵     } さきの 図2.38と同じ操作
218E:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B   ~...@..`........
218E:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60   @..`..... ..@..`
218E:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02   ..... ..@..`....
218E:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B   ! .#@.%`.'..) .+
218E:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60   @.-`./..1 .3@.5`
218E:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04   .7..9 .;@.=`.?..
218E:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B   Ap.C@.E`.G..I .K
218E:0070  C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 60   @.M`.O..Q .S@.U`
218E:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 FF 0F 06   .W..Y .[@.]`....
218E:0090  61 20 06 63 40 06 65 60-06 67 80 06 69 A0 06 6B   a .c@.e`.g..i .k
218E:00A0  C0 06 6D E0 06 6F 00 07-71 20 07 73 40 07 75 60   @.m`.o..q .s@.u`
218E:00B0  07 77 F0 FF 00 00 00 00-00 00 00 00 00 00 00 00   .wp.............
218E:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

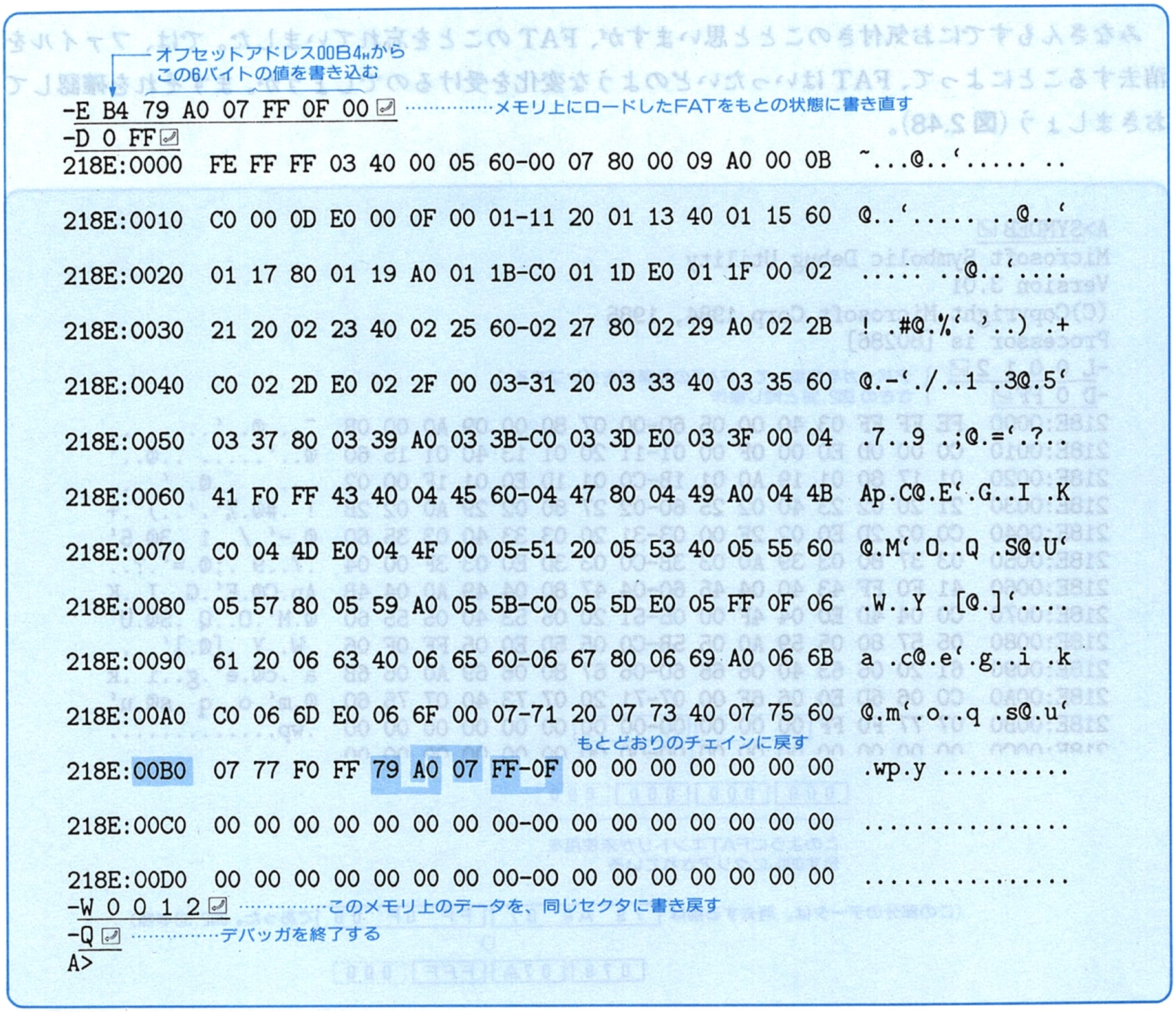
このようにFATエントリが未使用を示す000Hにクリアされている

（この部分のデータは、消去する前は 79 A0 07 FF 0F 00 であった。図2.38参照）

図 2.48　消去されたファイルの FAT エントリを見る

このように、消去されたファイルが使用していたクラスタに対応するFATエントリが、すべて000Hになっています。FATエントリが000Hの場合は、それに対応するクラスタが未使用であることを意味しています。ファイルを作成したり、書き込んだりするときに、MS-DOSはFATエントリが000Hであるクラスタを捜し出すことによって、空きクラスタを見つけ出すということを思い出してください。ファイルを消去すれば、そのファイルが格納されていたクラスタに対応するFATエントリは000Hになるのです。

結局、ファイルを復活させるには、このFATエントリも、もとの状態に戻さなければなりません。この実験では、もとの状態がわかっているので、これをもとのとおりに書き換えてみましょう(図2.49)。



```
-E B4 79 A0 07 FF 0F 00⏎
-D 0 FF⏎
218E:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B  ~...@..‘..... ..
218E:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60  @..‘..... ..@..‘
218E:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02  ..... ..@..‘....
218E:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B  ! .#@.%‘.’..) .+
218E:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60  @.-‘./..1 .3@.5‘
218E:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04  .7..9 .;@.=‘.?..
218E:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B  Ap.C@.E‘.G..I .K
218E:0070  C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 60  @.M‘.O..Q .S@.U‘
218E:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 FF 0F 06  .W..Y .[@.]‘....
218E:0090  61 20 06 63 40 06 65 60-06 67 80 06 69 A0 06 6B  a .c@.e‘.g..i .k
218E:00A0  C0 06 6D E0 06 6F 00 07-71 20 07 73 40 07 75 60  @.m‘.o..q .s@.u‘
218E:00B0  07 77 F0 FF 79 A0 07 FF-0F 00 00 00 00 00 00 00  .wp.y ..........
218E:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
218E:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-W 0 0 1 2⏎
-Q⏎
A>
```

オフセットアドレス00B4Hから
この6バイトの値を書き込む

……………メモリ上にロードしたFATをもとの状態に書き直す

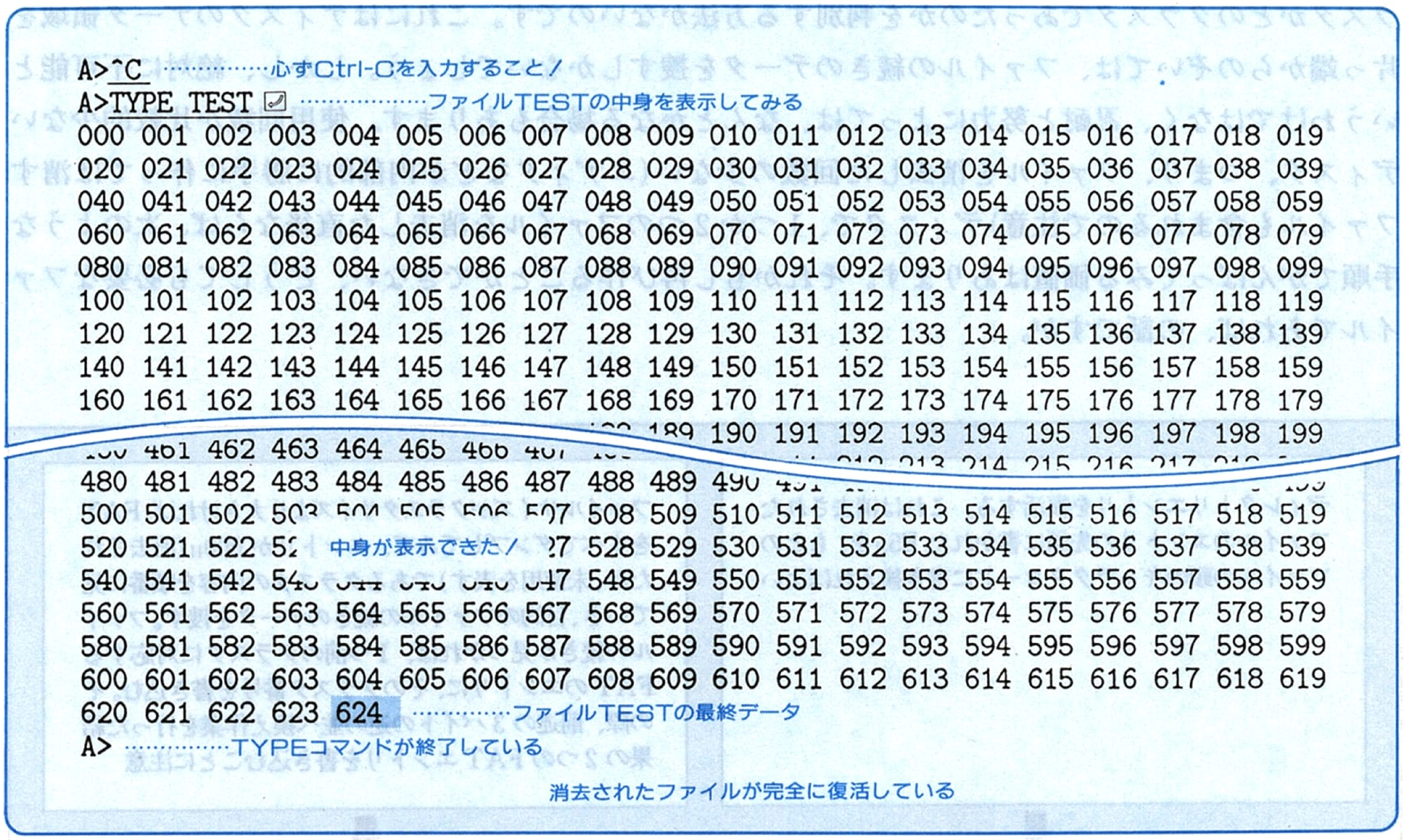
もとどおりのチェインに戻す

……………このメモリ上のデータを、同じセクタに書き戻す

……………デバッガを終了する

図2.49　FATエントリを消去前の状態に書き直す

以上の作業で、消去されたファイルTESTに対応していたFATエントリをもとの状態に書き直すことができました。再度TYPEコマンドでその内容を確認してみましょう（**図2.50**）。



```
A>^C
A>TYPE TEST
000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019
020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039
040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059
060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079
080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
                                        189 190 191 192 193 194 195 196 197 198 199
    461 462 463 464 465 466
480 481 482 483 484 485 486 487 488 489 490
500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519
520 521 522 52                 27 528 529 530 531 532 533 534 535 536 537 538 539
540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559
560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579
580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599
600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619
620 621 622 623 624
A>
```

……………必ずCtrl-Cを入力すること！

……………ファイルTESTの中身を表示してみる

中身が表示できた！

……………ファイルTESTの最終データ

……………TYPEコマンドが終了している

消去されたファイルが完全に復活している

図2.50　FATエントリを復活させたファイルTESTを表示する

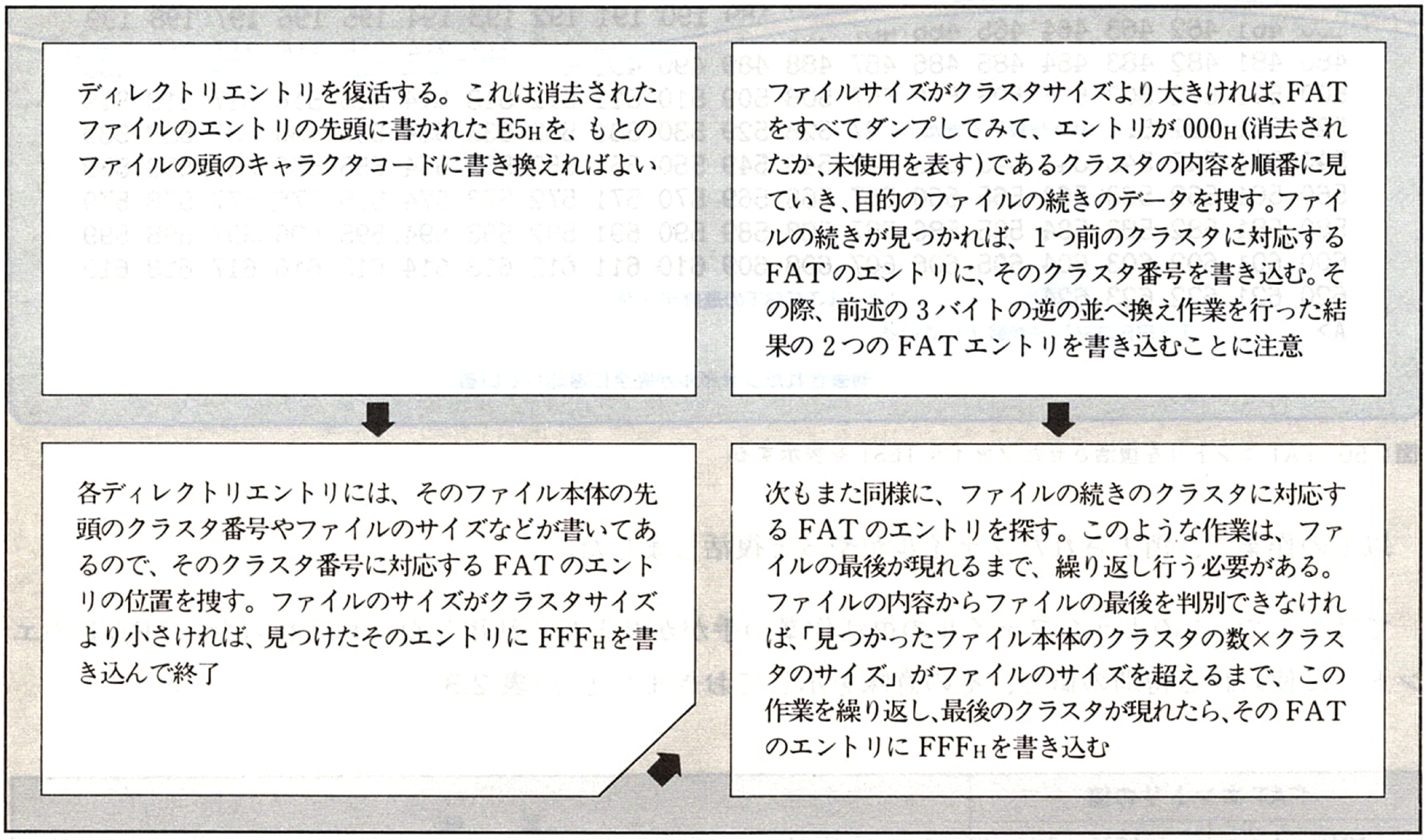
以上の作業で、消去されたファイルがやっと復活しました。

ではここで、このようなファイルの復活作業の手がかりとして活用しなければならない、FATのエントリに使われる特別の値と、その意味を示しておきましょう（**表2.3**）。

FATエントリの値		意　味
12ビット	16ビット	
000H	0000H	対応するクラスタは未使用
001H	0001H	この値は使われない
002H～FF6H	0002H～FFF6H	対応するクラスタは使用中であり、この値は次のクラスタ番号を示している
FF7H	FFF7H	対応するクラスタ内にスキップセクタがある
FF8H～FFFH	FFF8H～FFFFH	対応するクラスタはファイル本体、あるいはサブディレクトリ本体の最後のクラスタであることを示す（実際にはFFFHまたはFFFFHのみが使われている）

表2.3　FATエントリの特別な値とその意味

さて、ここで行った、消去したファイルを復活させる作業は、消去する以前のディレクトリやFATの状態がわかっているので比較的簡単に成功しました。しかし実際には、一般にFATエントリを復活するのは容易ではありません。むしろ非常に困難です。最初のクラスタを見つけたとしても、次のクラスタがどのクラスタであったのかを判別する方法がないのです。これにはディスクのデータ領域を片っ端からのぞいては、ファイルの続きのデータを捜すしかないでしょう。しかし、絶対に不可能というわけではなく、忍耐と努力によっては、なんとかなる場合もあります。使用回数が比較的少ないディスク、つまり、ファイルを消去した回数の少ない(エディタなどが内部的に勝手に作っては消すファイルも含まれるので注意)ディスクで、1つか2つのファイルを消去した直後ならば、次のような手順でがんばってみる価値はあります。それがもし再び作ることができない、どうしても必要なファイルであれば、の話ですが。



ディレクトリエントリを復活する。これは消去されたファイルのエントリの先頭に書かれたE5Hを、もとのファイルの頭のキャラクタコードに書き換えればよい

各ディレクトリエントリには、そのファイル本体の先頭のクラスタ番号やファイルのサイズなどが書いてあるので、そのクラスタ番号に対応するFATのエントリの位置を捜す。ファイルのサイズがクラスタサイズより小さければ、見つけたそのエントリにFFFHを書き込んで終了

ファイルサイズがクラスタサイズより大きければ、FATをすべてダンプしてみて、エントリが000H(消去されたか、未使用を表す)であるクラスタの内容を順番に見ていき、目的のファイルの続きのデータを捜す。ファイルの続きが見つかれば、1つ前のクラスタに対応するFATのエントリに、そのクラスタ番号を書き込む。その際、前述の3バイトの逆の並べ換え作業を行った結果の2つのFATエントリを書き込むことに注意

次もまた同様に、ファイルの続きのクラスタに対応するFATのエントリを探す。このような作業は、ファイルの最後が現れるまで、繰り返し行う必要がある。ファイルの内容からファイルの最後を判別できなければ、「見つかったファイル本体のクラスタの数×クラスタのサイズ」がファイルのサイズを超えるまで、この作業を繰り返し、最後のクラスタが現れたら、そのFATのエントリにFFFHを書き込む

注：ここには12ビットFATの場合を示す

以上で復活作業は終わりです。うまく行けば、消去されたファイルが復活しているでしょう。とくに使用回数の少ないディスクの場合は、連続したクラスタに連続してファイルが割り当てられているのが普通なので、復活できる可能性は比較的高くなります*。

---

* この作業を、半自動的に行ってくれるツール、たとえば「エコロジー」などが市販されている。

2.3.11　サブディレクトリ内の消去されたファイルの復活

今までは、消去されたファイルがルートディレクトリにある場合について考えてきました。では、サブディレクトリ内のファイルを消去した場合はどうすればよいのでしょう。この場合の手順も、ルートディレクトリでの場合に比べて、とくにむずかしくなるわけではありません。ルートディレクトリを対象としていたのが、サブディレクトリになるだけです。

サブディレクトリは、データ領域に格納されており、ファイル本体と同じ「ファイル」として扱われていますので、サブディレクトリを見るには、今までのファイルの中身を見たときと同じ手順で行います。サブディレクトリのディレクトリエントリを捜すには、今までのファイル名の欄にサブディレクトリ名が書かれていますので、ルートディレクトリを見れば目的のサブディレクトリか、もしくはその親ディレクトリが存在していることがわかります。また、そのエントリのファイル属性の欄にはディレクトリであることを示す10Hが書かれていますので、それが通常のファイルのエントリではなくサブディレクトリのエントリであることが確認できます。

そして、ルートディレクトリ内の目的のサブディレクトリ(またはその親ディレクトリ)のエントリに書かれているクラスタ番号から、サブディレクトリの本体がどこにあるかがわかります。また、その本体に対応するFATエントリを見れば、それに続くサブディレクトリの本体(複数のクラスタが使われている場合)のクラスタがわかります。ただし、サブディレクトリのサイズ欄は必ず00Hになっていますので、サブディレクトリの本体が何クラスタを占有しているかの判断はできません。ですから、サブディレクトリの最後を知るには、通常のファイルのときに行ったように、ディレクトリエントリの先頭が00Hであるエントリ(エントリの最終を示す)を目印にするか、FATエントリがFFFHである(1つのファイル、またはサブディレクトリの最終クラスタを示す)クラスタを目印にするかして、その間をたどらなければなりません。

では、サブディレクトリ上のファイルTESTを消去してみましょう。このファイルは、さきほどの実行例で使用したテキストファイルTESTと同じもので、これを新しいシステムディスク上にサブディレクトリを作成してそこにコピーしたものです(**図2.51**参照)。実行例を**図2.52**に示します。ここで行われている操作を順に追ってみてください。

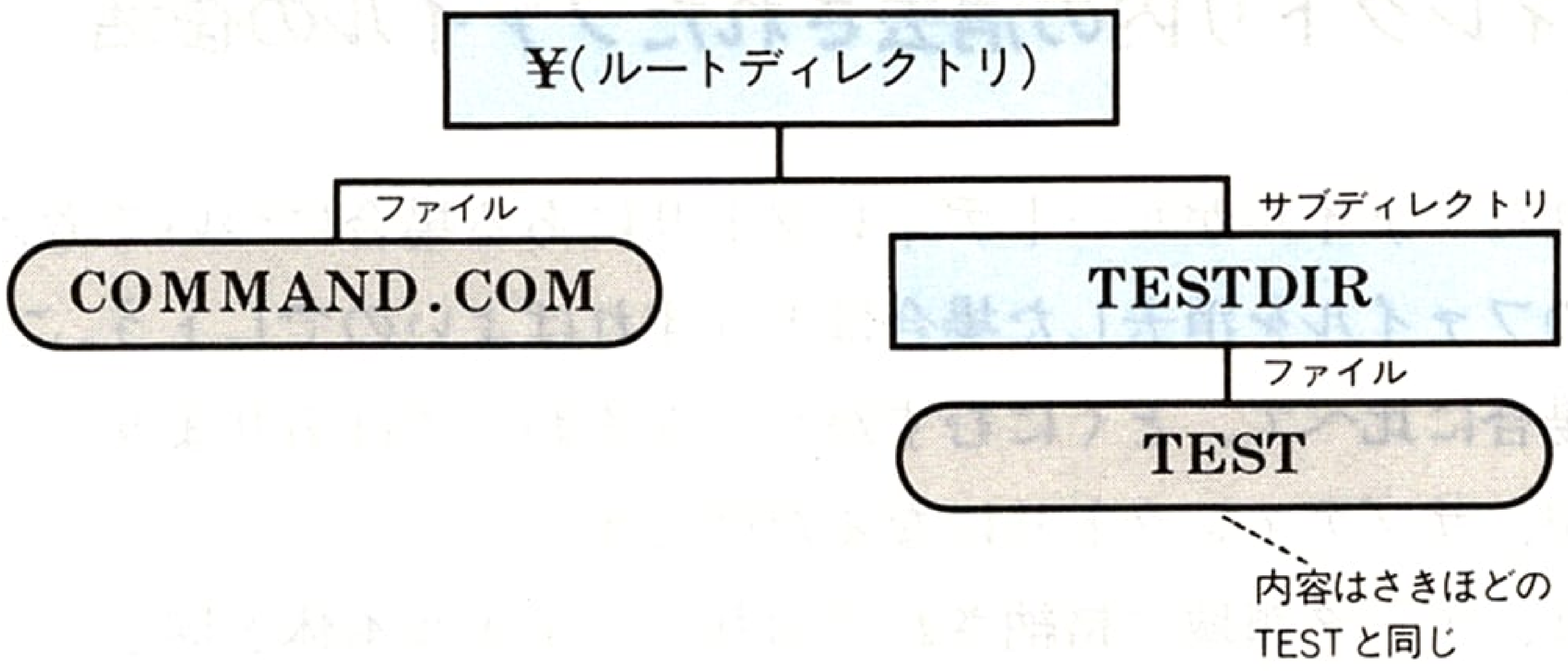

¥(ルートディレクトリ)
ファイル
サブディレクトリ
COMMAND.COM
TESTDIR
ファイル
TEST
内容はさきほどの
TEST と同じ

図 2.51　サブディレクトリ上のファイルが消去された場合の復活実験に使用するディスクの階層構造

```
A>DIR↵ ……………ここでの復活実験に使うディスクのルートディレクトリのファイルを見る

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13    0:00
TESTDIR      <DIR>      89-08-22    8:42 ……………サブディレクトリTESTDIRがある
        2 個のファイルがあります.
  1125376 バイトが使用可能です.

A>CD TESTDIR↵ ……………カレントディレクトリをTESTDIRに移す

A>DIR↵ ……………そこのファイルを見る

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥TESTDIR

.            <DIR>      89-08-22    8:42
..           <DIR>      89-08-22    8:42
TEST              2500  89-08-22    5:17 ……………カレントディレクトリTESTDIRには、
        3 個のファイルがあります.             ファイルTESTのみ存在している
  1125376 バイトが使用可能です.

A>DEL TEST↵ ……………ファイルTESTを消去する

A>DIR↵ ……………その確認

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥TESTDIR

.            <DIR>      89-08-22    8:42
..           <DIR>      89-08-22    8:42
        2 個のファイルがあります. ←――――消去されている
  1128448 バイトが使用可能です.
```

— 図 2.52 — (次ページ以下に続く)

```
A>SYMDEB⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-L 0 0 5 1⏎
-D 0 FF⏎
20D4:0000  49 4F 20 20 20 20 20 20-53 59 53 27 00 00 00 00  IO      SYS'....
20D4:0010  00 00 00 00 00 00 02 00-ED 10 02 00 00 00 01 00  ........m.......
20D4:0020  4D 53 44 4F 53 20 20 20-53 59 53 27 00 00 00 00  MSDOS   SYS'....
20D4:0030  00 00 00 00 00 00 02 00-ED 10 42 00 40 72 00 00  ........m.B.@r..
20D4:0040  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00  COMMAND COM ....
20D4:0050  00 00 00 00 00 00 02 00-ED 10 5F 00 63 61 00 00  ........m._.ca..
20D4:0060  54 45 53 54 44 49 52 20-20 20 20 10 00 00 00 00  TESTDIR    .....
20D4:0070  00 00 00 00 00 00 4E 45-16 13 78 00 00 00 00 00  ......NE..x.....
```

ルートディレクトリの先頭部の論理セクタ番号(5インチ2HDのディスクの場合)

デバッガを起動して、このディスクのルートディレクトリの先頭部をダンプする

サブディレクトリ名TESTDIR

ファイルの属性はディレクトリ

サブディレクトリTESTDIRの
ディレクトリエントリ

サブディレクトリTESTDIRのディレクトリ本体の先頭は、
クラスタ番号078$_H$のクラスタに格納されている

次はサブディレクトリを見る

```
-L 0 0 81 1⏎
-D 0 FF⏎
20D4:0000  2E 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  .          .....
20D4:0010  00 00 00 00 00 00 4E 45-16 13 78 00 00 00 00 00  ......NE..x.....
20D4:0020  2E 2E 20 20 20 20 20 20-20 20 20 10 00 00 00 00  ..         .....
20D4:0030  00 00 00 00 00 00 4E 45-16 13 00 00 00 00 00 00  ......NE........
20D4:0040  E5 45 53 54 20 20 20 20-20 20 20 00 00 00 00 00  eEST        ....
20D4:0050  00 00 00 00 00 00 2D 2A-16 13 79 00 C4 09 00 00  ......-*..y.D...
```

サブディレクトリ本体の論理 セクタ番号
5インチ2HDのディスクの場合　0B$_H$＋(78$_H$−2)×1＝81$_H$

サブディレクトリTESTDIRのディレクトリ本体の先頭部をダンプする

自分自身のディレクトリ

親ディレクトリ

消去されたので空きマークとなっている

ファイルTESTの本体は、
クラスタ番号79$_H$のクラスタ
にあった

消去されたファイル
TESTのディレク
トリであった箇所

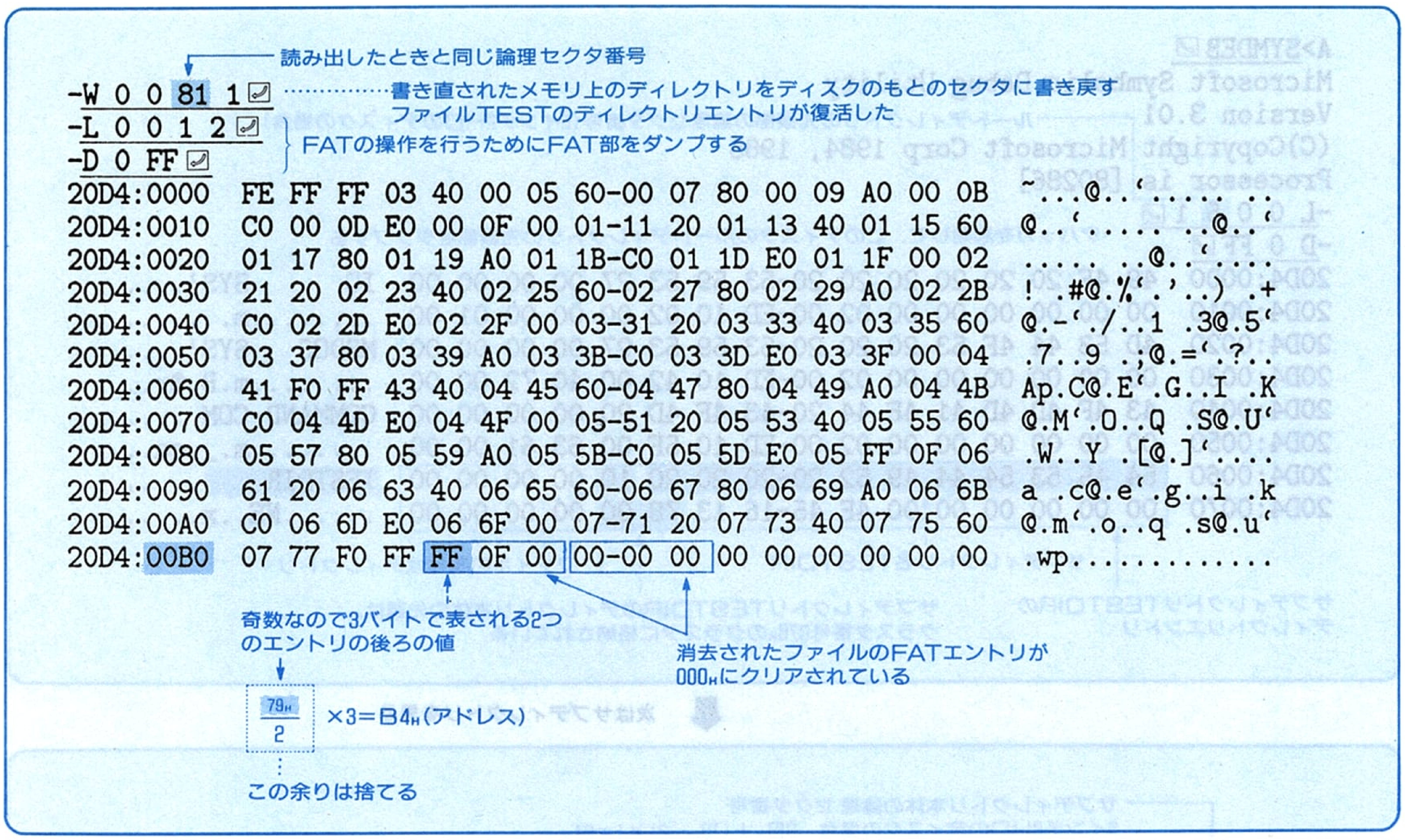
次はメモリ上の空きマーク(E5$_H$)をもとのデータに書き直す

```
-E 40 54⏎
-D 0 FF⏎
20D4:0000  2E 20 20 20 20 20 20 20-20 20 20 10 00 00 00 00  .          .....
20D4:0010  00 00 00 00 00 00 4E 45-16 13 78 00 00 00 00 00  ......NE..x.....
20D4:0020  2E 2E 20 20 20 20 20 20-20 20 20 10 00 00 00 00  ..         .....
20D4:0030  00 00 00 00 00 00 4E 45-16 13 00 00 00 00 00 00  ......NE........
20D4:0040  54 45 53 54 20 20 20 20-20 20 20 00 00 00 00 00  TEST        ....
20D4:0050  00 00 00 00 00 00 2D 2A-16 13 79 00 C4 09 00 00  ......-*..y.D...
```

空きマークE5$_H$に書き換えられた箇所を、もとのデータ「T」(54$_H$)に書き直す

もとのデータ(ファイル名の先頭文字)に書き直された

次は書き直したディレクトリをもとのセクタに書き戻す



```
-W 0 0 81 1⏎
-L 0 0 1 2⏎
-D 0 FF⏎
20D4:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B   ~...@..‘......
20D4:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60   @..‘..... ..@..‘
20D4:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02   ..... ..@..‘....
20D4:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B   ! .#@.%‘.’..) .+
20D4:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60   @.-‘./..1 .3@.5‘
20D4:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04   .7..9 .;@.=‘.?..
20D4:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B   Ap.C@.E‘.G..I .K
20D4:0070  C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 60   @.M‘.O..Q .S@.U‘
20D4:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 FF 0F 06   .W..Y .[@.]‘....
20D4:0090  61 20 06 63 40 06 65 60-06 67 80 06 69 A0 06 6B   a .c@.e‘.g..i .k
20D4:00A0  C0 06 6D E0 06 6F 00 07-71 20 07 73 40 07 75 60   @.m‘.o..q .s@.u‘
20D4:00B0  07 77 F0 FF FF 0F 00 00-00 00 00 00 00 00 00 00   .wp.............
```

読み出したときと同じ論理セクタ番号

書き直されたメモリ上のディレクトリをディスクのもとのセクタに書き戻す
ファイルTESTのディレクトリエントリが復活した

FATの操作を行うためにFAT部をダンプする

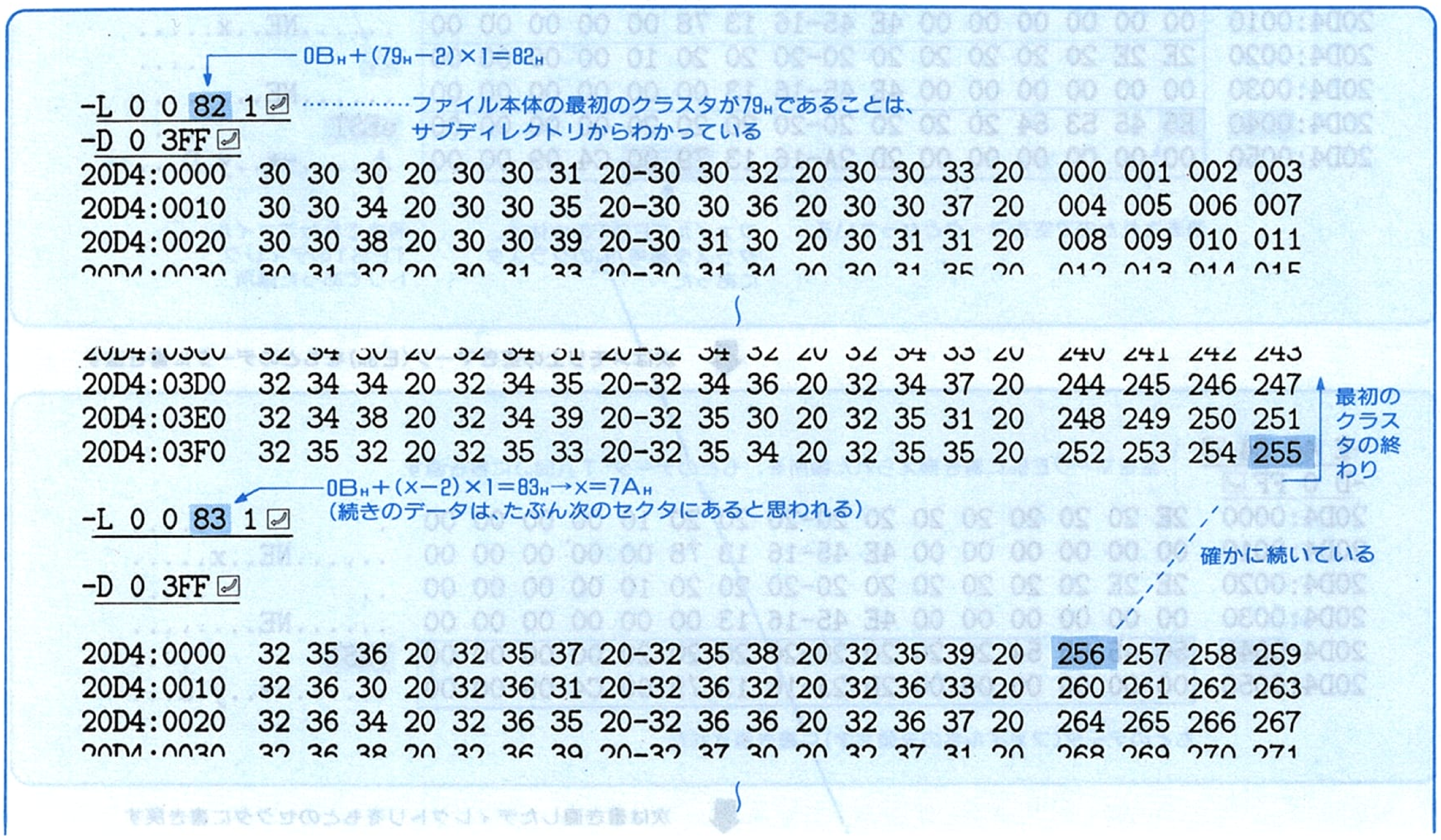
奇数なので3バイトで表される2つのエントリの後ろの値

消去されたファイルのFATエントリが000ₕにクリアされている

$\frac{79_H}{2}$ ×3＝B4ₕ(アドレス)

この余りは捨てる

次は消去されたこのFATエントリ(クラスタのチェイン情報）を推定して、その内容を確認しながら決定する作業。これが最もたいへんである。



```
-L 0 0 82 1⏎
-D 0 3FF⏎
20D4:0000  30 30 30 20 30 30 31 20-30 30 32 20 30 30 33 20   000 001 002 003
20D4:0010  30 30 34 20 30 30 35 20-30 30 36 20 30 30 37 20   004 005 006 007
20D4:0020  30 30 38 20 30 30 39 20-30 31 30 20 30 31 31 20   008 009 010 011
20D4:0030  30 31 32 20 30 31 33 20-30 31 34 20 30 31 35 20   012 013 014 015
                               ⁝
20D4:03C0  32 34 30 20 32 34 31 20-32 34 32 20 32 34 33 20   240 241 242 243
20D4:03D0  32 34 34 20 32 34 35 20-32 34 36 20 32 34 37 20   244 245 246 247
20D4:03E0  32 34 38 20 32 34 39 20-32 35 30 20 32 35 31 20   248 249 250 251
20D4:03F0  32 35 32 20 32 35 33 20-32 35 34 20 32 35 35 20   252 253 254 255
-L 0 0 83 1⏎

-D 0 3FF⏎

20D4:0000  32 35 36 20 32 35 37 20-32 35 38 20 32 35 39 20   256 257 258 259
20D4:0010  32 36 30 20 32 36 31 20-32 36 32 20 32 36 33 20   260 261 262 263
20D4:0020  32 36 34 20 32 36 35 20-32 36 36 20 32 36 37 20   264 265 266 267
20D4:0030  32 36 38 20 32 36 39 20-32 37 30 20 32 37 31 20   268 269 270 271
```

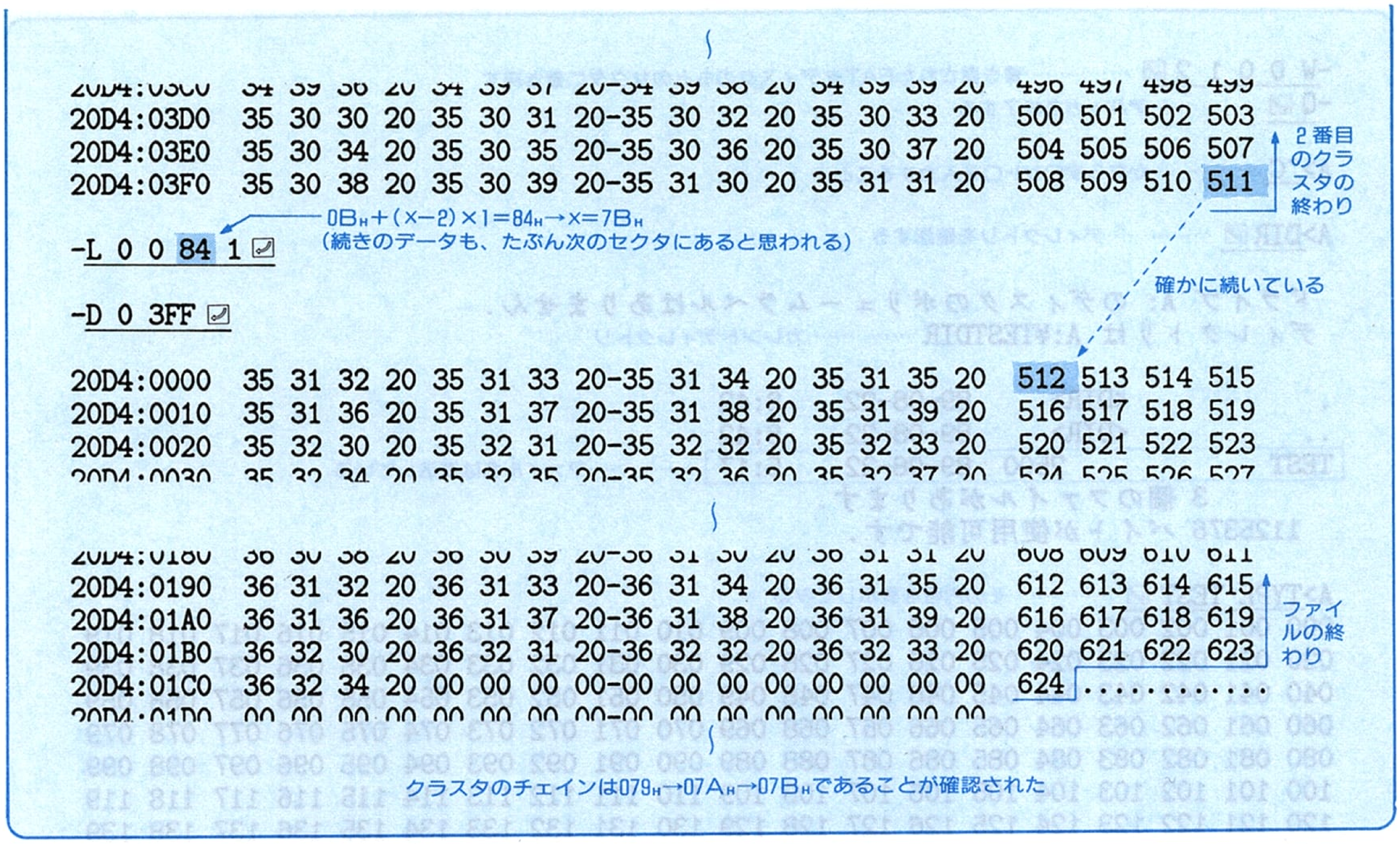
0Bₕ＋(79ₕ－2)×1＝82ₕ

ファイル本体の最初のクラスタが79ₕであることは、サブディレクトリからわかっている

最初のクラスタの終わり

0Bₕ＋(x－2)×1＝83ₕ→x＝7Aₕ
(続きのデータは、たぶん次のセクタにあると思われる)

確かに続いている



```
                              ∫
20D4:03C0  34 39 36 20 34 39 37 20-34 39 38 20 34 39 39 20  496 497 498 499
20D4:03D0  35 30 30 20 35 30 31 20-35 30 32 20 35 30 33 20  500 501 502 503
20D4:03E0  35 30 34 20 35 30 35 20-35 30 36 20 35 30 37 20  504 505 506 507
20D4:03F0  35 30 38 20 35 30 39 20-35 31 30 20 35 31 31 20  508 509 510 511

-L 0 0 84 1⏎

-D 0 3FF⏎

20D4:0000  35 31 32 20 35 31 33 20-35 31 34 20 35 31 35 20  512 513 514 515
20D4:0010  35 31 36 20 35 31 37 20-35 31 38 20 35 31 39 20  516 517 518 519
20D4:0020  35 32 30 20 35 32 31 20-35 32 32 20 35 32 33 20  520 521 522 523
20D4:0030  35 32 34 20 35 32 35 20-35 32 36 20 35 32 37 20  524 525 526 527
                              ∫
20D4:0180  36 30 38 20 36 30 39 20-36 31 30 20 36 31 31 20  608 609 610 611
20D4:0190  36 31 32 20 36 31 33 20-36 31 34 20 36 31 35 20  612 613 614 615
20D4:01A0  36 31 36 20 36 31 37 20-36 31 38 20 36 31 39 20  616 617 618 619
20D4:01B0  36 32 30 20 36 32 31 20-36 32 32 20 36 32 33 20  620 621 622 623
20D4:01C0  36 32 34 20 00 00 00 00-00 00 00 00 00 00 00 00  624............
20D4:01D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
                              ∫
```

2番目のクラスタの終わり

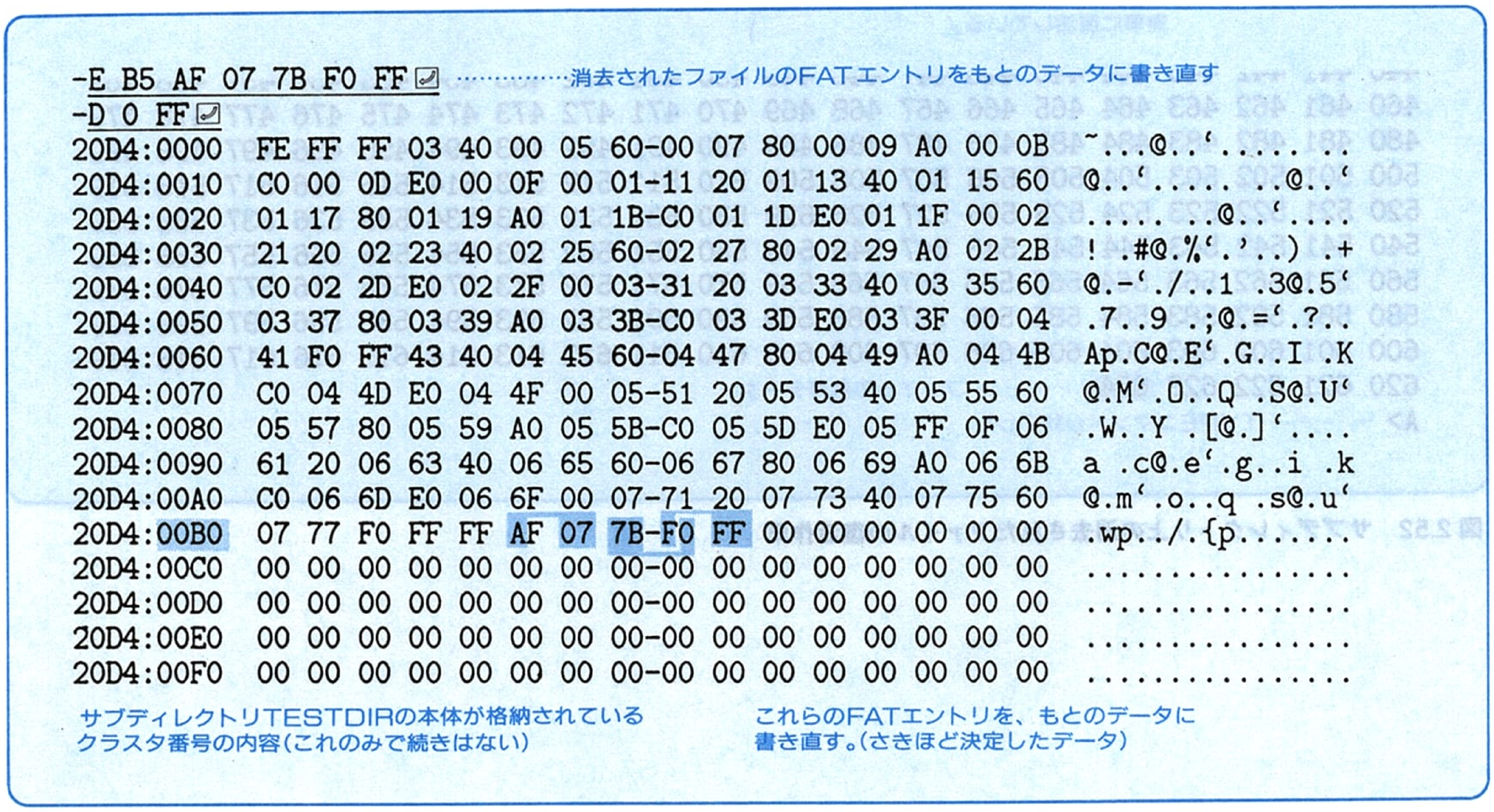
$0B_H+(x-2)\times1=84_H \rightarrow x=7B_H$
（続きのデータも、たぶん次のセクタにあると思われる）

確かに続いている

ファイルの終わり

クラスタのチェインは$079_H \rightarrow 07A_H \rightarrow 07B_H$であることが確認された

次はこのクラスタのチェイン情報によって
FATエントリを復元する



```
-E B5 AF 07 7B F0 FF⏎
-D 0 FF⏎
20D4:0000  FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B  ~...@..‘.......
20D4:0010  C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60  @..‘..... ..@..‘
20D4:0020  01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02  ..... ..@..‘....
20D4:0030  21 20 02 23 40 02 25 60-02 27 80 02 29 A0 02 2B  ! .#@.%‘.’..) .+
20D4:0040  C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60  @.-‘./..1 .3@.5‘
20D4:0050  03 37 80 03 39 A0 03 3B-C0 03 3D E0 03 3F 00 04  .7..9 .;@.=‘.?..
20D4:0060  41 F0 FF 43 40 04 45 60-04 47 80 04 49 A0 04 4B  Ap.C@.E‘.G..I .K
20D4:0070  C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 60  @.M‘.O..Q .S@.U‘
20D4:0080  05 57 80 05 59 A0 05 5B-C0 05 5D E0 05 FF 0F 06  .W..Y .[@.]‘....
20D4:0090  61 20 06 63 40 06 65 60-06 67 80 06 69 A0 06 6B  a .c@.e‘.g..i .k
20D4:00A0  C0 06 6D E0 06 6F 00 07-71 20 07 73 40 07 75 60  @.m‘.o..q .s@.u‘
20D4:00B0  07 77 F0 FF FF AF 07 7B-F0 FF 00 00 00 00 00 00  .wp../.{p......
20D4:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
20D4:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
20D4:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
20D4:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

……………消去されたファイルのFATエントリをもとのデータに書き直す

サブディレクトリTESTDIRの本体が格納されている
クラスタ番号の内容（これのみで続きはない）

これらのFATエントリを、もとのデータに
書き直す。（さきほど決定したデータ）

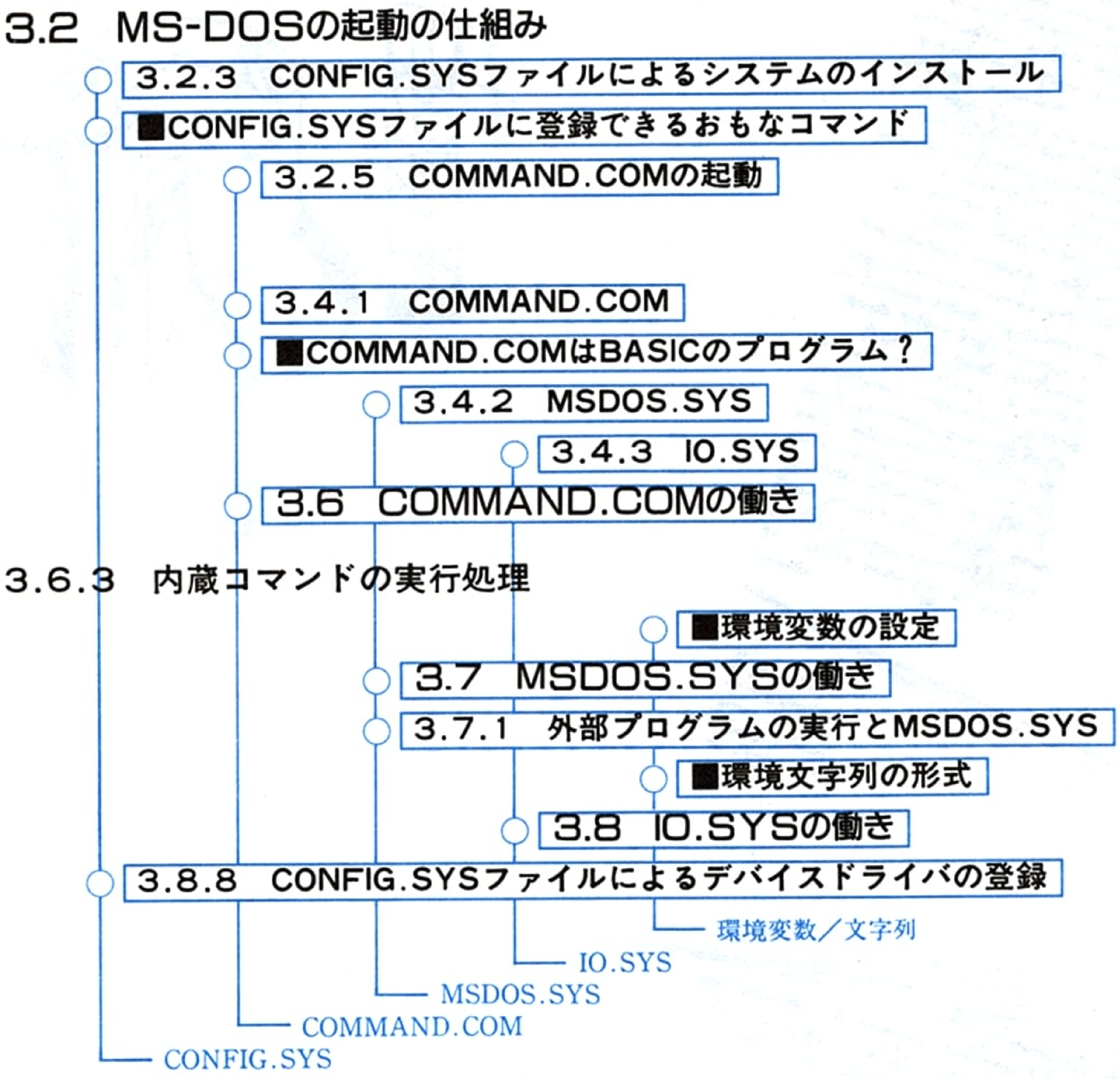
次はもとの状態に書き直した
FATをディスクに書き戻す

```
-W 0 0 1 2↵ ……………書き直されたFATをディスクのもとのセクタに書き戻す
-Q↵ ……………デバッガを終了する

A>^C ……………かならずCtrl-Cを入力すること！

A>DIR↵ ……………ディレクトリを確認する

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥TESTDIR ……………カレントディレクトリ

.            <DIR>     89-08-22     8:42
..           <DIR>     89-08-22     8:42
TEST             2500  89-08-22     5:17  ……………ファイル名は復活している
        3 個のファイルがあります.
  1125376 バイトが使用可能です.

A>TYPE TEST↵ ……………その内容を表示してみる
000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019
020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039
040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057 058 059
060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079
080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179

                  無事に復活している！

440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459
460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479
480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499
500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519
520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539
540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559
560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579
580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599
600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619
620 621 622 623 624 ……………ファイルの最終データ
A> ……………TYPEコマンドが終了した
```

図 2.52　サブディレクトリ上の消去されたファイルの復活作業

3章
MS-DOSの仕組みと働き

本章では、MS-DOSのシステム構成について解説しましょう。MS-DOSは、起動する前はディスク上の3つの独立したプログラムとして存在しています。みなさんもよくご存知の「IO.SYS」「MSDOS.SYS」「COMMAND.COM」の3つのシステムファイルがそれです。これらがコンピュータのメインメモリにロードされると、互いに連携をとりながらOS(オペレーティングシステム)としての働きを開始します。ところで、これらはなぜ3つに分離されているのでしょうか。きっと、それぞれ異なった役割を持っているに違いありません。本章は、これら各部の役割を解説しながら、MS-DOSの仕組みと各部の働きを明らかにしていきましょう。


3.2 MS-DOSの起動の仕組み
3.2.3 CONFIG.SYSファイルによるシステムのインストール
■CONFIG.SYSファイルに登録できるおもなコマンド
3.2.5 COMMAND.COMの起動
3.4.1 COMMAND.COM
■COMMAND.COMはBASICのプログラム？
3.4.2 MSDOS.SYS
3.4.3 IO.SYS
3.6 COMMAND.COMの働き
3.6.3 内蔵コマンドの実行処理
■環境変数の設定
3.7 MSDOS.SYSの働き
3.7.1 外部プログラムの実行とMSDOS.SYS
■環境文字列の形式
3.8 IO.SYSの働き
3.8.8 CONFIG.SYSファイルによるデバイスドライバの登録
環境変数／文字列
IO.SYS
MSDOS.SYS
COMMAND.COM
CONFIG.SYS

本章も2章に似て、各テーマが散在していますので、それらのおもな項目の関連図を示しておきます。各テーマについて互いに参照しながら読み進むとよいでしょう(ただし、この図がすべての関連を網羅しているわけではありません)。

3.1 OS(オペレーティングシステム)とは

MS-DOSとは、「MicroSoft-Disk Operating System」のことであり、「DiskOS」といっているのは、このOSの主たる機能が、ディスク管理であることを意味しています。オペレーティングシステムとは何かについては『入門MS-DOS』でも述べていますが、ここではもう少し専門的に解説してみましょう。MS-DOSの内部構造を理解するためにも、まずOSというものの概念を知ることが先決です。1章、2章と読み進み、実習してこられたみなさんは、あるいはその概要をすでにつかんでいるかもしれません。

3.1.1 OSのないコンピュータシステムを使うと

ではまず、なぜOSというものが生まれたのか、その成立過程を考えるために、OSがまったく存在しない状態のコンピュータを想定してみましょう。つまり、MS-DOSはもちろん、BASICなども組み込まれておらず、簡単なモニタ(BASIC内のモニタコマンドによる各種の機能と同じような働きをするプログラム)も載っていない、ソフトウェア的には何も用意されていない状態のシステムを考えてください。そのシステムのハードウェア構成は、入出力装置としてキーボードとディスプレイ、それに外部記憶装置としてフロッピーディスクドライブなどが接続されている一般的なシステムとしましょう。

さてみなさんは、このシステム上で実行するための、あるプログラムを開発しなければならない立場に置かれています。ただし、プログラムを開発するためには、開発環境の整ったMS-DOSマシンなどのシステムを別途使うことにします。しかし、その開発マシンで作成されたプログラムを実際に利用するのは、OSに類する基本ソフトウェアがまったく用意されていないコンピュータシステムなのです。まずはこの前提をよく頭に入れておいてください。ここではその2つのシステムを、「開発マシン」および「実行マシン」と呼ぶことにしましょう(**図3.1**)。

さて、基本ソフトウェアが用意されていないシステム上で実行するプログラムは、どのような作り方をすればよいのでしょう。それはかなりたいへんな作業になることを覚悟しなければなりません。まず、最も基本的なルーチンである、キーボードから入力された1文字のデータを取り出すプログラムから作成しなければならないでしょう(**リスト3.1**参照)。次は任意の1文字をディスプレイに表示するプログラムを作成しなければなりません。これらはいずれも最も基本的なルーチンですが、これらの単純な機能のプログラムを組むにも、キーボードやディスプレイの入出力ポート(それらとコンピュータ本体との接点)に対するアクセス法を知らなければなりません。これには、各装置の入出力に関するハード/ソフト両面の知識が必要になります。

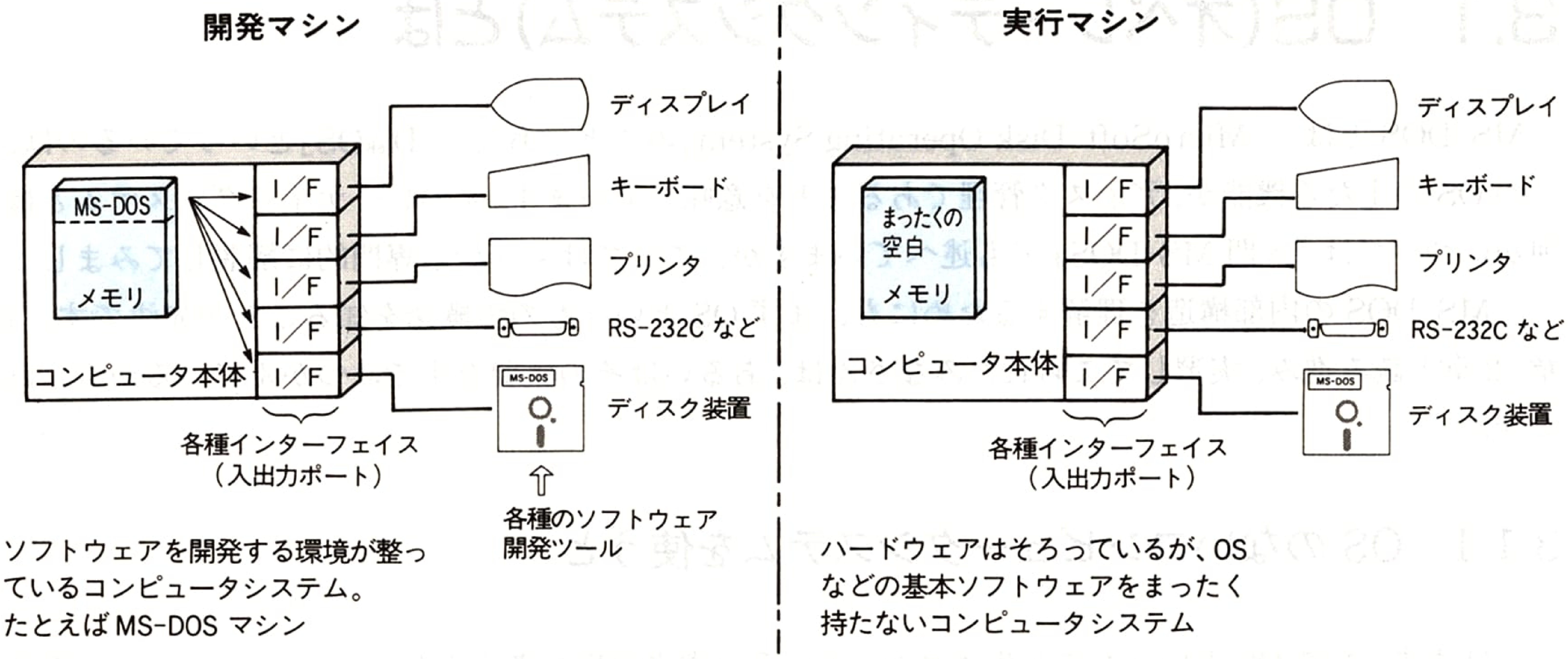

開発マシン
MS-DOS
メモリ
コンピュータ本体
I／F
I／F
I／F
I／F
I／F
ディスプレイ
キーボード
プリンタ
RS-232C など
MS-DOS
ディスク装置
各種インターフェイス
（入出力ポート）
各種のソフトウェア
開発ツール
ソフトウェアを開発する環境が整っているコンピュータシステム。
たとえばMS-DOS マシン
実行マシン
まったくの
空白
メモリ
コンピュータ本体
I／F
I／F
I／F
I／F
I／F
ディスプレイ
キーボード
プリンタ
RS-232C など
MS-DOS
ディスク装置
各種インターフェイス
（入出力ポート）
ハードウェアはそろっているが、OSなどの基本ソフトウェアをまったく持たないコンピュータシステム

図 3.1　開発マシンと実行マシン

作成するプログラムの中でも、とくにコンピュータシステムを動かす部分のプログラムの作成は、まったくのゼロからの手作りであり、たいへんな苦労があると思います。ここではとりあえず途中の過程を省略して、目的のプログラムが開発マシン上でめでたく完成したとしましょう。完成したプログラムは、開発マシンのフロッピーディスク上にセーブされています。

ではそのプログラムを、どのようにして実行マシンにロードすればよいのでしょうか。実行マシンにもフロッピーディスクドライブが付いているのだからディスクから読み込ませれば？ どうやって？ あたりまえですがそれは不可能です。ディスクは付いていても、そのディスクを動かすプログラムはどこにもありません。

では、フロッピーディスク上のプログラムファイルを読み出して、メモリ上にロードするプログラムを作ればよい…確かにそうです。しかしディスクの読み出しプログラムを作るには、まずフロッピーディスクドライブを制御するための心臓部であるFDC（フロッピーディスク・コントロール用のLSI）の機能と使い方に関するハード／ソフト両面の知識が必要です。ところがこの知識があっても、ただ任意のセクタを読み書きするプログラムが作れるだけであり、ここで必要とする、ディスク上に「ファイル」として格納されている完成したプログラムファイルを読み出すプログラムは作れません。2章を読まれたみなさんは、ファイルを読み出すプログラムを作成するのは**ファイルシステム**を作り上げることにほかならず、まあ普通では無理だということが理解できるでしょう。

ではフロッピーディスクからデータをロードするのはあきらめて、開発マシンからRS-232Cの通信インターフェイスでデータを受信したらどうだろうか？ これはよいアイデアです。その通信プログラムを作るのは、少なくてもディスクファイルを読み出すプログラムよりずっと簡単です。

```
;        KEY INPUT SAMPLE 1/20 BY M.F.
;
MOCW2           EQU     00H
KEY_DATA        EQU     41H
KEY_STATUS      EQU     43H
KEY_COMMAND     EQU     43H
;
SYSTEM_SEG      SEGMENT AT 0
                ORG     9*4
KEYINT_OFF      DW      ?
KEYINT_SEG      DW      ?
SYSTEM_SEG      ENDS
;
SUB1    SEGMENT
        ASSUME  CS:SUB1,DS:SYSTEM_SEG
;
KEY_INIT:
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DS
        CLI
        MOV     AX,SYSTEM_SEG
        MOV     DS,AX
        MOV     AX,SUB1
        XCHG    AX,KEYINT_SEG
        MOV     CS:SAVE_SEG,AX
        MOV     AX,OFFSET KEYINT_ENTRY
        XCHG    AX,KEYINT_OFF
        MOV     CS:SAVE_OFF,AX
;
        MOV     CX,8
        MOV     AL,80H
        MOV     BX,OFFSET KEY_TABLE
L1:     MOV     CS:[BX],AL
        INC     BX
        LOOP    L1
        POP     DS
        POP     CX
        POP     BX
        POP     AX
        STI
;
        RET
;
SAVE_SEG        DW      ?
SAVE_OFF        DW      ?
;
KEYINT_ENTRY:
        PUSH    AX
        PUSH    BX
        PUSH    DX
        PUSH    DI
;
        MUL     AX
        IN      AL,KEY_STATUS
        AND     AL,38H
        MOV     AL,16H
        OUT     KEY_COMMAND,AL
        IN      AL,KEY_DATA
        JZ      NEXT
        MOV     BX,OFFSET KEY_TABLE
        MOV     CX,8
        MOV     DL,AL
        OR      AL,AL
        JNS     MAKE1
        AND     AL,7FH
BREAK1:
        CMP     AL,CS:[BX]
        JZ      NEXT1
        INC     BX
        LOOP    BREAK1
        JMP     SHORT NEXT
MAKE1:
        TEST    CS:[BX],BYTE PTR 80H
        JNZ     NEXT1
        INC     BX
        LOOP    MAKE1
        JMP     SHORT NEXT
;
NEXT1:  MOV     CS:[BX],DL
;
NEXT:
        MOV     AL,20H
        OUT     MOCW2,AL
        POP     DI
        POP     DX
        POP     BX
        POP     AX
;
        IRET
;
KEY_TABLE       DB      8 DUP(?)
;
SUB1    ENDS
        END
```

キーボードからの1文字入力ルーチンなど簡単だと思われるが、多少とも気の利いた実用的なものとなると、この例のように、けっこうたいへんなプログラムとなる(このプログラム例は、「PC-9801マシン語入門」(アスキー出版局刊)より引用)

リスト 3.1　キーボードから 1 文字入力するルーチンの例

その受信プログラムは、あまり苦労もなく開発マシン上ででき上がったとしましょう。このプログラムをどうにかして実行マシンにロードしなければなりません。あれ?! また同じことになりました。でも今回の受信プログラムは小さいプログラムなので、そのマシン語を1バイトずつメモリに書き込んでいってもよさそうです。そのためには、まずキーボードから入力したデータを、1バイトずつメモリに書き込むプログラムを作ろう…。

この話について、これ以上の深入りはやめておきましょう。MS-DOSもない、BASICもない、基本ソフトウェアが何にもないコンピュータがどのようなものか、ここで想定した例でも明らかでしょう。いずれにしても、基本ソフトウェアのないマシン上で実行するプログラムを作成するには、その実行マシンとなるコンピュータ本体や、各種周辺装置のハードウェアのすべての部分について詳しく知っていなければならず、またかなりのソフトウェア技術力を必要とするのです。(図3.2参照)

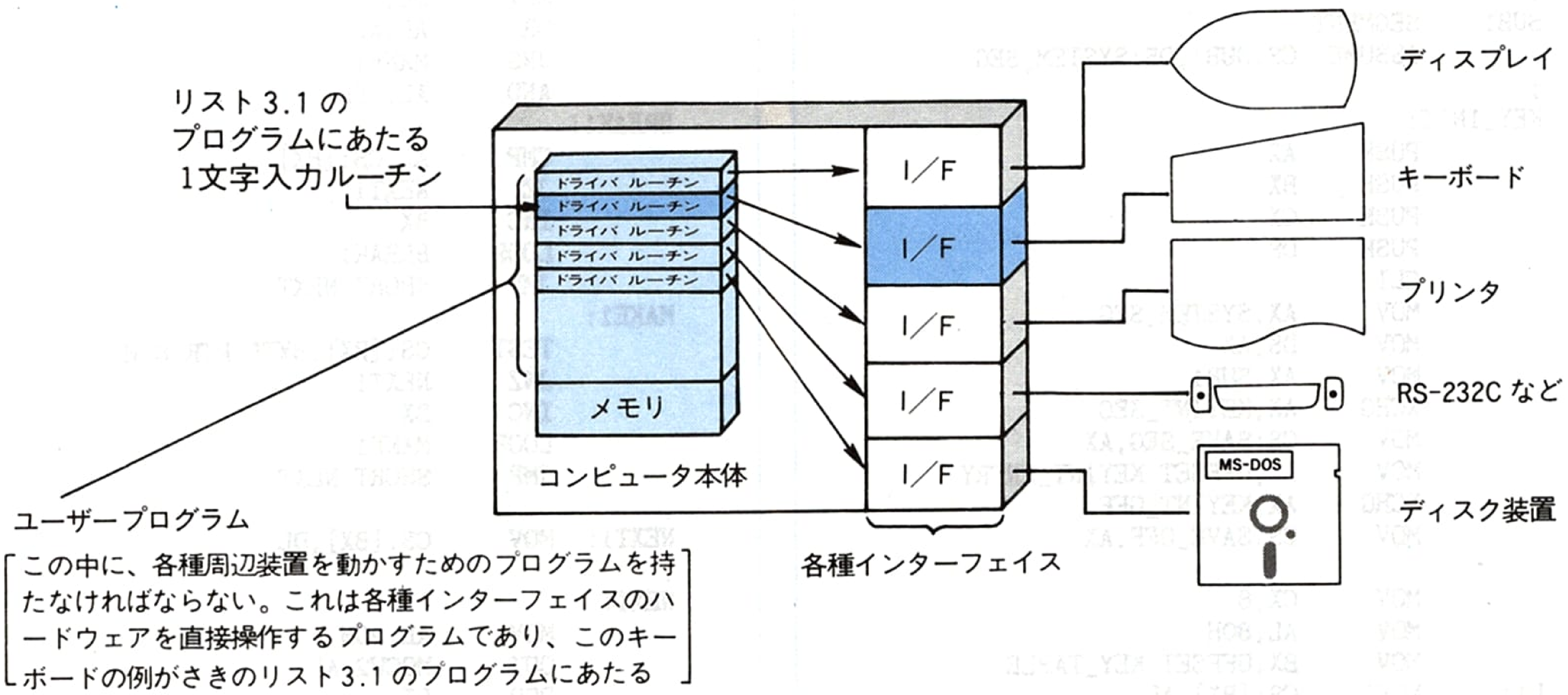

リスト3.1の
プログラムにあたる
1文字入力ルーチン
ドライバ ルーチン
ドライバ ルーチン
ドライバ ルーチン
ドライバ ルーチン
ドライバ ルーチン
メモリ
コンピュータ本体
I／F
I／F
I／F
I／F
I／F
各種インターフェイス
ディスプレイ
キーボード
プリンタ
RS-232Cなど
MS-DOS
ディスク装置
ユーザープログラム
この中に、各種周辺装置を動かすためのプログラムを持たなければならない。これは各種インターフェイスのハードウェアを直接操作するプログラムであり、このキーボードの例がさきのリスト3.1のプログラムにあたる

図3.2　ハードウェアをユーザープログラムで直接操作する

3.1.2　OSの概念の誕生

さて、あなたが優秀なコンピュータ技術者であり、ハードウェアにもソフトウェアにも強く、ここで想定している実行マシン上で実行する、いくつかのプログラムが開発できたとしましょう。そのいくつかのプログラムの内部では、キーボードから1文字入力する部分や、ディスプレイに1文字出力する部分などのルーチンに、きっと同じものを使い回していることでしょう。また、いろいろなプログラムを開発している間には、次第に高度なプログラムも手掛けるようになり、ディスクとの簡単な形式での入出力もできるようになっているかもしれません(図3.3参照)。

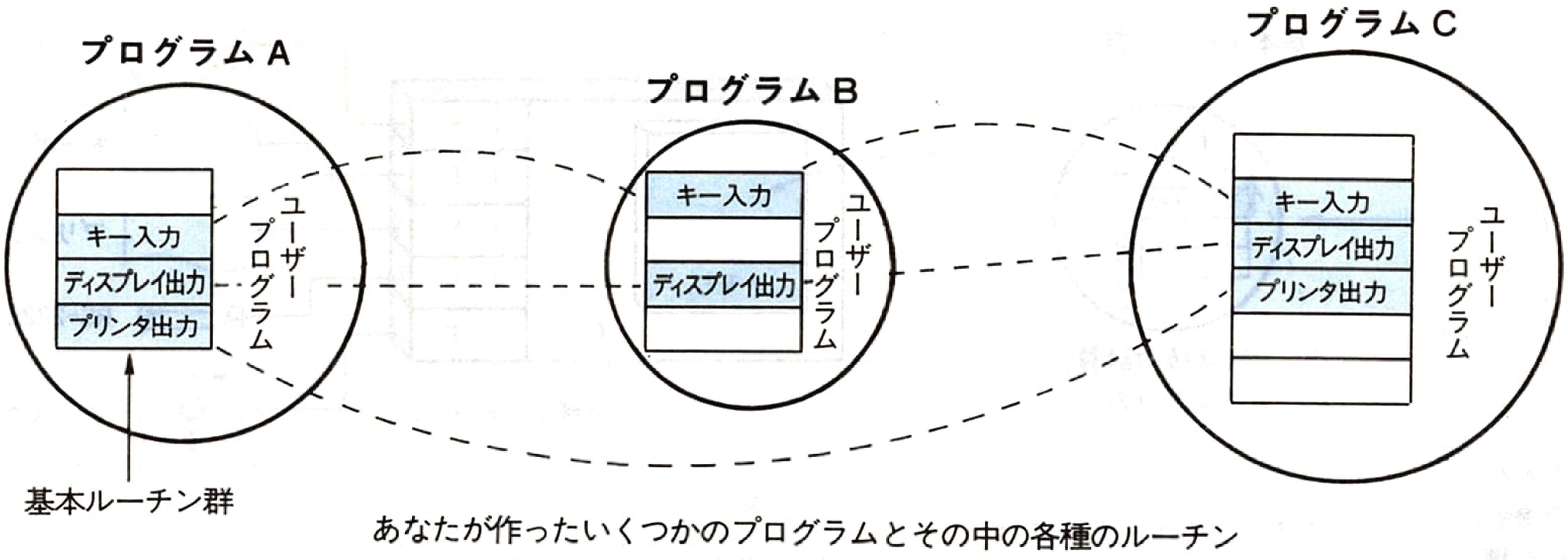


図 3.3　基本的なルーチンは各プログラムで共用される

さあこれからが大切なところです。あなたのほかにも、実行マシンとまったく同じ構成のコンピュータシステム上で実行するための、別のプログラムを開発している人、A さんがいるとしましょう。たとえば、あなたが科学技術計算のプログラムを作り、A さんはビジネスプログラムを作っているとします。でき上がったこの2つのプログラムを比較してみると、A さんが作ったビジネスプログラムの中にも、部分的には、きっとあなたとほとんど同じようなルーチンが使われているはずです。たとえば、キーボードから1文字入力したり、ディスプレイに1文字出力したりするルーチンは、ほとんど同じプログラムでしょう。しかし、ディスクの入出力などのルーチンは、たとえばあなたのプログラムでは、簡単なファイル形式のシーケンシャル読み出し／書き込みが可能であり、A さんのプログラムでは、セクタ単位の直接読み出し／書き込みができるだけ、というように、基本機能からまったく異なっているかもしれません（**図 3.4** 参照）。

後日この2人が、同じ型の実行マシンのために作成した、それぞれのプログラムについて話し合う機会があったとしましょう。「あそこのルーチンはこんなテクニックを使っているんだ」などと話しているうちに、お互いに、「この部分のルーチンは君の作ったプログラムの方が使いやすくてスマートだ」とか、「それと同じ機能のルーチンを作ろうとしているんだけれど、むずかしくてまだ未完成なんだ」などという話になるでしょう。

こうなると、互いに作ったプログラムを利用し合わない手はありません。2人が賢明なら、互いのプログラムの中で共有できそうな部分を整理して、共用できる形式に直すでしょう。また、新たに作らなければならないルーチンでも、それが互いに必要であるなら、その作業を分担することにより、開発の効率を上げることもできます。

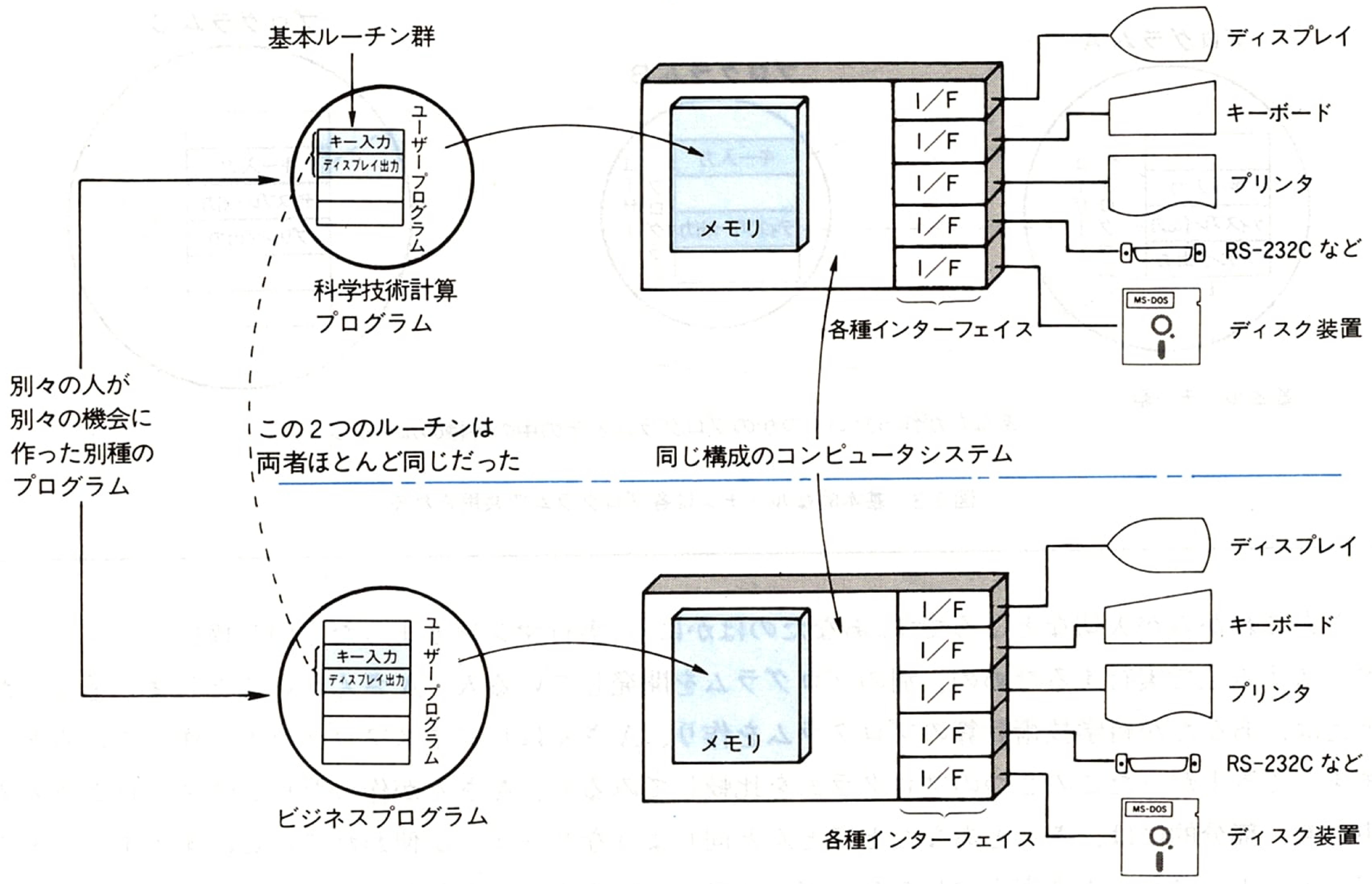

基本ルーチン群
キー入力
ディスプレイ出力
ユーザープログラム
科学技術計算プログラム
別々の人が別々の機会に作った別種のプログラム
この2つのルーチンは両者ほとんど同じだった
メモリ
I/F
ディスプレイ
キーボード
プリンタ
RS-232Cなど
MS-DOS
ディスク装置
各種インターフェイス
同じ構成のコンピュータシステム
キー入力
ディスプレイ出力
ユーザープログラム
ビジネスプログラム
メモリ
I/F
ディスプレイ
キーボード
プリンタ
RS-232Cなど
MS-DOS
ディスク装置
各種インターフェイス

図3.4　別々の人が作った2つのプログラムの中には、たいてい同じ機能のルーチンがある

3.1.3　ライブラリの誕生

上の話を発展させ、多くの人が共通に利用できるようにしたルーチンが**ライブラリ**と呼ばれるものにほかなりません。図書館(ライブラリ)では、誰もが読みたい本を借りて読むことができます。つまり、共通に使えるいろいろなルーチンを用意して、必要な部分を誰もが利用できるようにしたものをライブラリと呼ぶのです(**図3.5**)。

OSも一種のライブラリといえます。コンピュータシステムの基本動作を行わせ、ユーザープログラムを実行するための、誰もが必要とする最低限のプログラムを集めたものをOSの基本と考えてもよいでしょう。ただし、最近ではかなり豊富で高度な機能を備えていないとOSとは呼べないほど、ハード/ソフトともに進歩していますが。

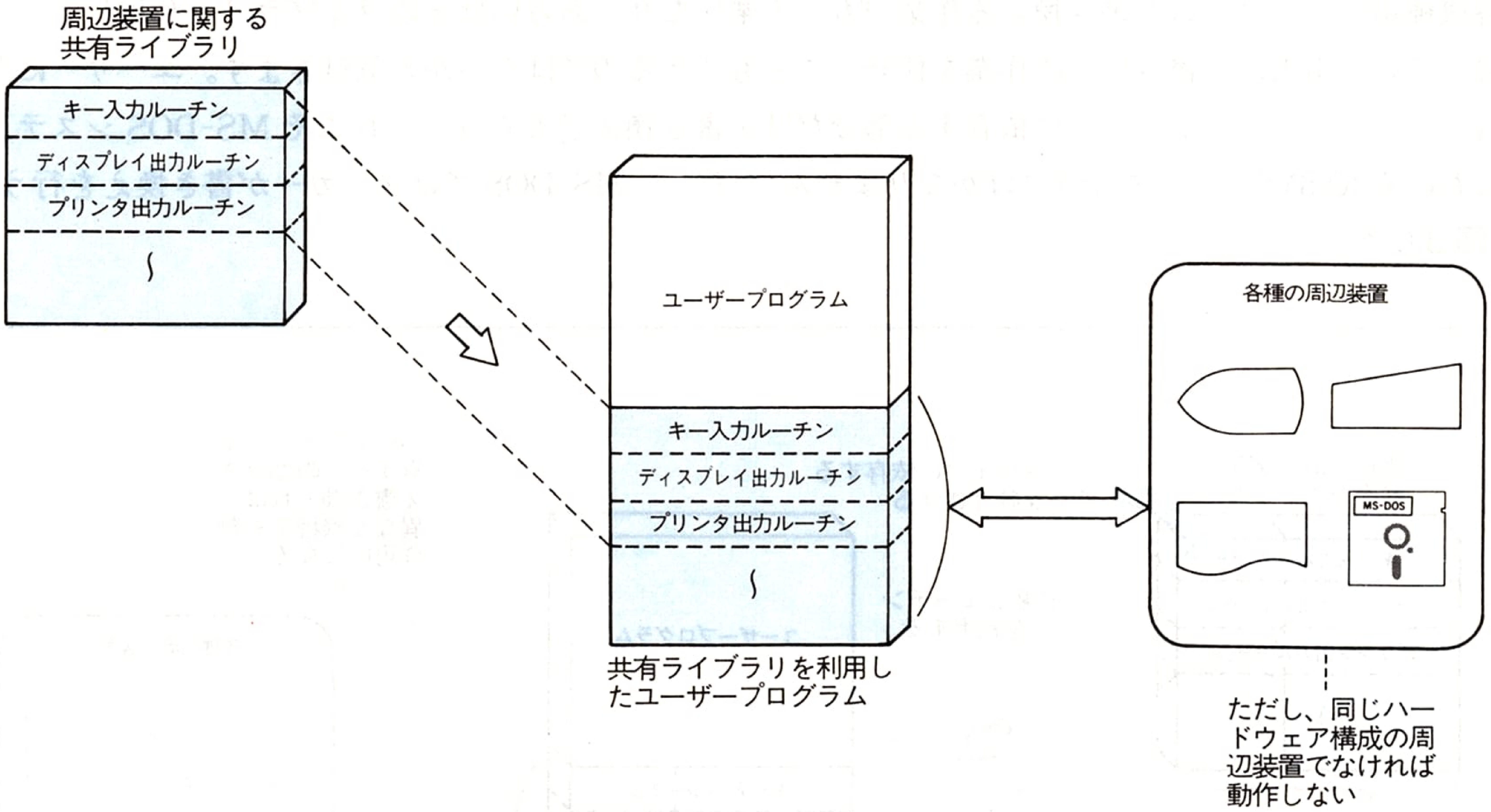

周辺装置に関する
共有ライブラリ
キー入力ルーチン
ディスプレイ出力ルーチン
プリンタ出力ルーチン
ユーザープログラム
キー入力ルーチン
ディスプレイ出力ルーチン
プリンタ出力ルーチン
共有ライブラリを利用したユーザープログラム
各種の周辺装置
MS-DOS
ただし、同じハードウェア構成の周辺装置でなければ動作しない

図 3.5　多くの人が共用できるルーチンを集めたライブラリ

3.1.4　ハードウェアに依存する部分の独立

さて、話を例のプログラムの開発にもどしましょう。あなたは多くのプログラムを作り、ライブラリも徐々に整ってきました。ディスクの読み／書きについては、2章で述べたようなファイルシステムの簡単なものを構築することに成功しているかもしれません。

そうこうしているとき、ハードウェアが異なる別のコンピュータシステムを使っている友達Ｂさんが、あなたが作ったファイルシステムなどのライブラリを、ぜひ自分のシステムで利用したいと言い出したとしましょう。ところが、あなたのライブラリは、あなたやＡさんの実行マシン専用のプログラムなので、機種の異なるコンピュータシステムでは、その一部を書き換えなければ動作しません。友達思いのあなたは、このプログラムを書き換えてあげます。

このようなことをしていると、またまた別なコンピュータシステムを使っている人が、ぜひ自分のシステムにも、ということにもなります。このように、いろいろな機種のために、自分の作ったプログラムを書き換える作業を行っているうちに、あなたは、多くのライブラリのほとんどについて、書き換えを必要とする部分は限られた一部であり、それはハードウェアに依存する箇所であることを発

見するでしょう。賢明なあなたは、ハードウェアに依存するその部分だけを分離して整理しておけば、各機種用にプログラムを書き換える作業がもっと楽になり、あるいはそのライブラリを利用しようと思っている本人に、書き換えの作業を任せることもできるのではないかと気付きます。ユーザーに各自のシステムのハードウェアに依存する部分だけを書き換えてもらう、これこそMS-DOSシステムにおける**IO.SYS**(後述)の部分にほかなりません(ただし、MS-DOSではメーカーが書き換えを行う)(図3.6参照)。

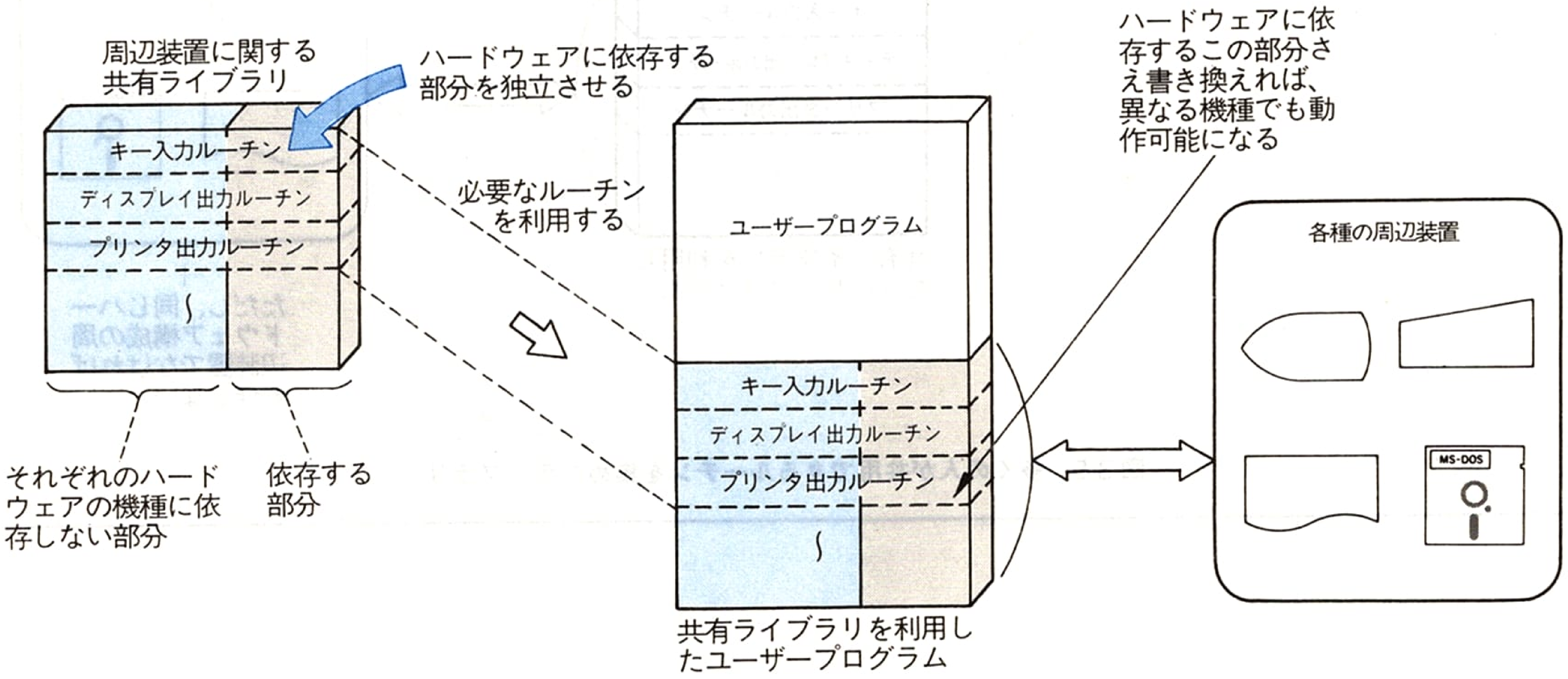

周辺装置に関する
共有ライブラリ
ハードウェアに依存する
部分を独立させる
キー入力ルーチン
ディスプレイ出力ルーチン
プリンタ出力ルーチン
必要なルーチン
を利用する
ユーザープログラム
ハードウェアに依
存するこの部分さ
え書き換えれば、
異なる機種でも動
作可能になる
各種の周辺装置
MS-DOS
それぞれのハード
ウェアの機種に依
存しない部分
依存する
部分
共有ライブラリを利用し
たユーザープログラム

図3.6　共有ライブラリ内のハードウェアに依存する部分を分離する

3.1.5　ユーザープログラムの各機種間での互換性

ユーザープログラムを作成する上で大切なことは、ハードウェアを操作するような部分(たとえばコンソールの入出力や、ディスクの読み出し／書き込みなど)はすべて共有ライブラリの中に含めておき、メインプログラムからハードウェアを操作する場合は、必ず共有ライブラリを介してアクセスするようにしておくことです。つまり、ハードウェアを直接操作する部分をメインプログラムから分離するのです。このように、すべての入出力を、ライブラリルーチンをコールすることによって行えば、メインプログラムはハードウェアに依存しなくなります(これが4章で解説する**システムコール**の機能にほかならない)。

このような考え方にすると、共有ライブラリ内のハードウェアに依存する部分のみ、各自のコンピュータシステムに合わせて書き換えれば、ユーザープログラムは異なる機種上でもそのまま実行することができることになります。

ここまでのことを要約すると、ユーザープログラムを作成するときには、まず、コンピュータシステムのハードウェアを操作する部分や、それらを組み合わせて基本的な入出力を行う部分を共有ライブラリとしてまとめ、それをメインプログラムから分離します。さらにその共有ライブラリから、ハードウェアを直接操作する部分を分離して、その部分を各機種ごとに用意します（図3.7）。このようにすれば、それぞれのユーザープログラムを書き換えることなく、異なる機種上で共通に実行できるのです。異なる機種間でもソフトウェアの互換性があるということは、実にすばらしいことです。

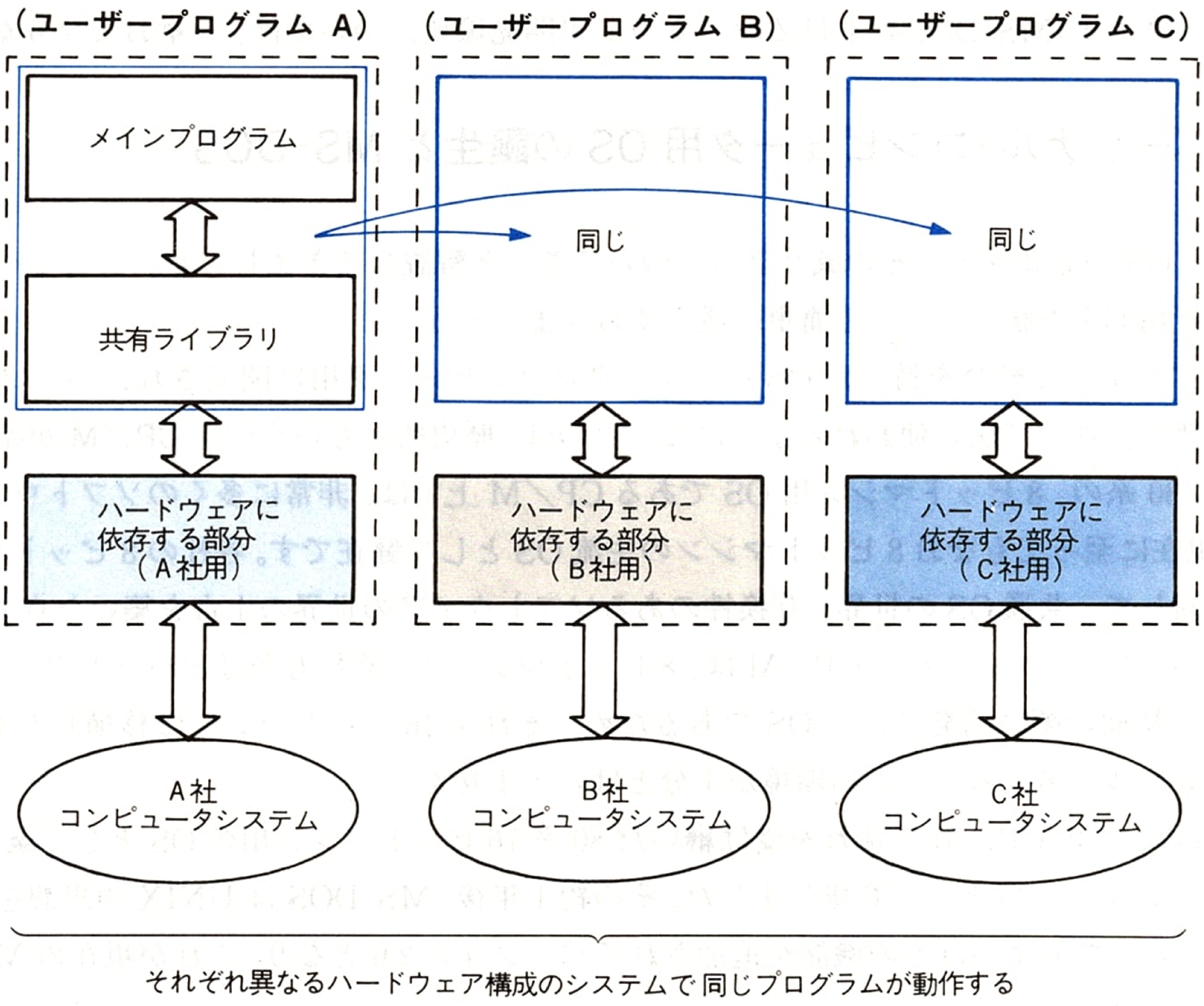

（ユーザープログラム A）
（ユーザープログラム B）
（ユーザープログラム C）
メインプログラム
共有ライブラリ
同じ
同じ
ハードウェアに依存する部分（A社用）
ハードウェアに依存する部分（B社用）
ハードウェアに依存する部分（C社用）
A社 コンピュータシステム
B社 コンピュータシステム
C社 コンピュータシステム
それぞれ異なるハードウェア構成のシステムで同じプログラムが動作する

図3.7　1つのユーザープログラムが異なる機種上で実行できる

3.1.6　ソフトウェア開発環境

OSの基本的な役割は、さきに述べたようにユーザープログラムが動作する環境を提供することにあります。パーソナル・コンピュータ用のOSは**DOS**と呼ばれているように、ディスクの管理（つまりファイル管理）が主体ですが、さらにもう1つの重要な役割として、ソフトウェアを開発するための環境の提供があります。みなさんは、UNIX（ユニックス）という名前を聞いたことがあると思います。UNIXはミニコン用のOSとして開発されたものですが、便利で使いやすいOSとしてたいへん注目されており、MS-DOSもバージョン2.0で、その思想を大幅に取り入れました。UNIXはまた、エンジニアリングワークステーション用のOSとしても注目されています。このUNIXが、別名「プログラマーのためのOS」と呼ばれるほど使いやすいといわれる理由は、プログラムを開発する環境が優れているからであり、このような完備したソフトウェア開発環境を提供することは、よいOSの基本的条件でもあるわけです（MS-DOSにおけるソフトウェア開発環境については、5章および6章参照）。

3.1.7　パーソナル・コンピュータ用OSの誕生とMS-DOS

これまで、OSの必要性と、その成り立ちについてざっと解説してきましたが、ここでパーソナル・コンピュータ用OSの歴史について簡単に述べておきましょう。

ここで述べたような概念を持ったOSが、マイクロコンピュータ用に開発され、パーソナル・コンピュータの普及に伴って広く使われるようになったのは、歴史的ともいうべき**CP／M**が始まりです。この8080、Z80系の、8ビットマシン用OSであるCP／M上では、非常に多くのソフトウェアが開発蓄積され、現在に至ってもなお8ビットマシンの主流OSとして健在です。各社の8ビットマシンの多くの機種に対して、共通OSの世界、互換性のあるソフトウェアの世界の土台を築いた点で、CP／Mはすばらしいものでした。しかしCP／Mは、8ビットマシンの、それも今ほどハードウェアが発達していない最も初期の頃に開発されたOSであるため、それを16ビットマシンに移植したCP／M-86は、現在のレベルで考えると、その環境が十分とはいえません。

MS-DOSは、このCP／Mの流れを受け継いだ80系16ビットマシン用のOSとして誕生し、私たちの前にバージョン1.25として登場しました。その約1年後、MS-DOSはUNIXの思想を取り入れ、新しいOSといってもよいほどの機能が追加されてバージョン2.0となり、これが現在のMS-DOSの基本となっています。

MS-DOSは、ソフトウェアを開発したり、その作成されたプログラムを実行するためのOSとして、柔軟かつ豊富な機能を持っています。本章では、このMS-DOSがどのような機能を持ち、その内部はどのような仕組みになっているかを、やさしく解説していきます。

3.1.8　MS-DOSの構造の予備知識

これまで、OSの成り立ちなどについて述べてきましたが、そのような理由から、MS-DOSは3つの部分に分かれており、その各部のプログラムがメモリ上に展開し、互いに連携してOSとしての働きをしています。

これらの3つの部分は、メモリ上にロードされる以前は、それぞれのプログラムファイルとしてディスク上に存在しています。2章でルートディレクトリのエントリデータを直接見たときの、**IO.SYS**と**MSDOS.SYS**、**COMMNAD.COM**というファイルがそれです。IO.SYSおよびMSDOS.SYSの2つは、システムファイルの属性が付けられているので、DIRコマンドでは表示されませんが、システムディスク（MS-DOSを起動できるもの）には必ず存在しています。MS-DOSのこの3つのファイルは、これまで述べてきたOSの話でのそれぞれの部分に、**図3.8**のように対応します。

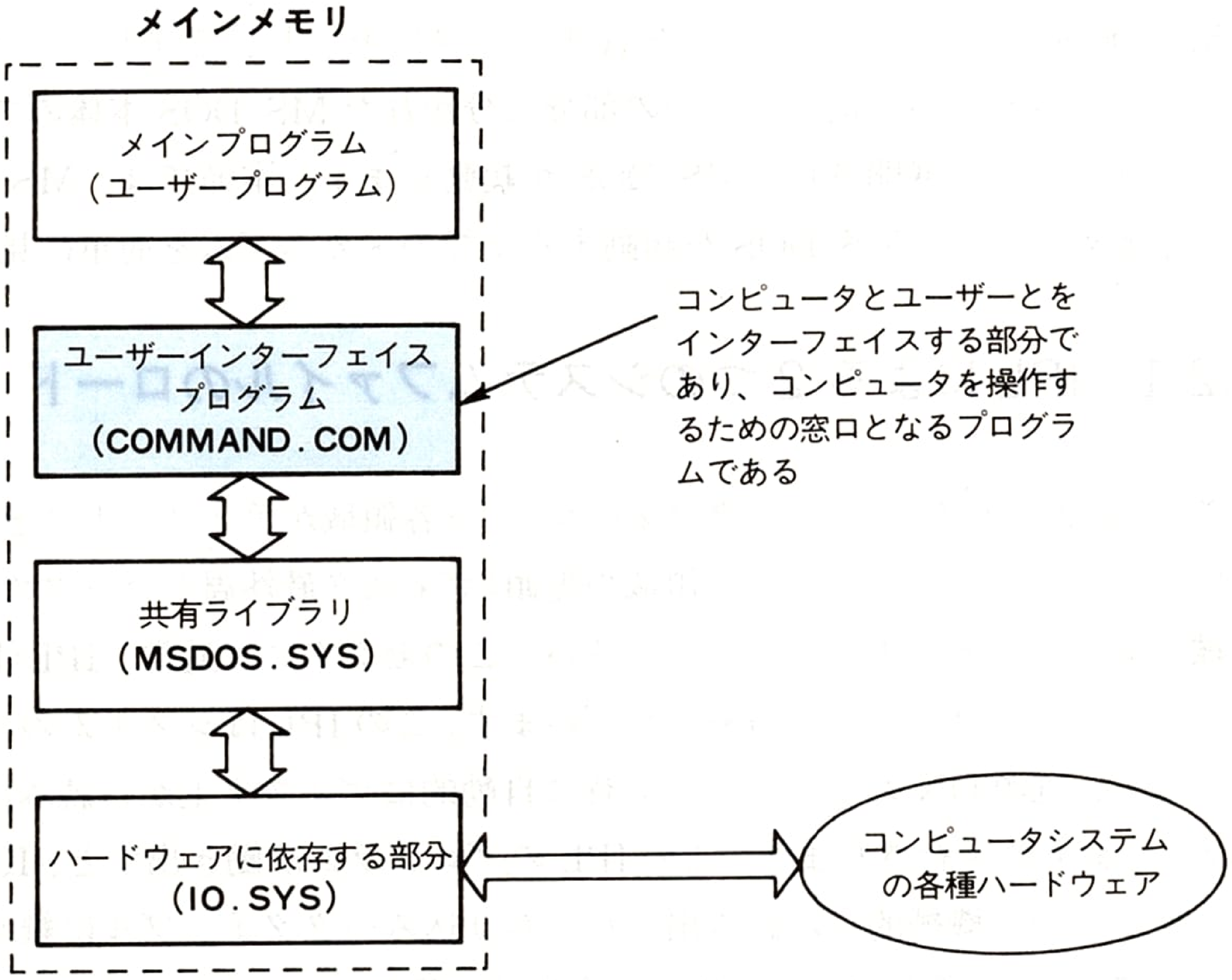

メインメモリ
メインプログラム
（ユーザープログラム）
ユーザーインターフェイス
プログラム
（COMMAND.COM）
共有ライブラリ
（MSDOS.SYS）
ハードウェアに依存する部分
（IO.SYS）
コンピュータシステム
の各種ハードウェア
コンピュータとユーザーとをインターフェイスする部分であり、コンピュータを操作するための窓口となるプログラムである

図3.8　MS-DOSの構成と、これまでのOSの話との対応

IO.SYSは、周辺装置のハードウェアを直接操作する低レベル(ハードウェアに近いほどレベルが低いと表現する)の入出力を行います。といっても多くのパーソナル・コンピュータの場合は、マシンに組み込みのROM内に、ハードウェアの基本的な操作を行うプログラム(BIOS：Basic Input／Output System)が用意されているため、周辺装置によっては、IO.SYSが直接ハードウェアを操作するわけではなく、このROM内BIOSとのインターフェイスをとるだけのものが多いようです。

MSDOS.SYSは、MS-DOSの本体であり、IO.SYSによる周辺装置の入出力機能を使って、コンソール入出力や、ディスクを管理するファイルシステムなどを実現します。

COMMAND.COMは、ユーザーからのコマンドを受け取り、その要求された意味を解釈して実行に移す部分です。DIR、COPY、TYPEなどの内蔵コマンドのプログラム本体は、このCOMMAND.COM内に組み込まれています。

3.2　MS-DOSの起動の仕組み

MS-DOSのシステムディスクをAドライブにセットしてリセットボタンを押すと、システムディスクに格納されている前述の3つの部分に分かれたMS-DOS本体のプログラムが、自動的に読み出されてメモリ上に展開され、MS-DOSが起動します。本節では、MS-DOSの内部構造を理解するための手始めとして、MS-DOSが起動するまでのメカニズムを簡単に見ていきましょう。

3.2.1　IPLによる2つのシステムファイルのロード

さて、2章の後半で、ディスクフォーマット(各領域がディスク上にどのように配置されているか)について述べましたが、ディスク領域の先頭(ディスク最外周トラックの最初のセクタ)は**システム予約領域**であったことを思い出してください。このセクタには通常、IPL(Initial Program Loader)と呼ばれる小さなプログラムが格納されています。このIPLはシステムのブート時(起動時)に必要なプログラムで、電源ONおよびリセット時に自動的にディスク上から読み出され、メモリ上にロードされたあと、直ちに実行されます。このIPLのプログラムが働き出すと、IO.SYSとMSDOS.SYSの2つのファイルが「機械的」に読み出され、割り込みベクタテーブルに続くメモリ上にロードされます。

ここでとくに「機械的」と表現した意味は、IPLのプログラムが、2章で解説したMS-DOSのファイルシステムに基づいたディスクのアクセス法で、IO.SYSとMSDOS.SYSを読み出すというわけではないからです。ファイルシステムのような複雑な仕事は、MS-DOSの心臓部であるMSDOS.SYSが行うものであり(この時点では、MS-DOS.SYSはまだメモリ上に存在していない)、IPLのような小さなプログラムが、実行できるしろものではありません。IPLは単純に、ディスク上の決まった場所

から、決まった数のセクタを読み出し、メモリ上にロードするだけの機能しかありません*(図 3.9)。

ということは、IO.SYS や MSDOS.SYS は、ディスク上の必ず決まった場所に置かれている必要があり、普通のファイルと同じように取り扱うわけにはいかないことになります。そのためこの2つのファイルは、ファイルの形態をとってはいますが、格納される領域は、データ領域の先頭から連続したセクタが固定的に割り当てられます。ただし、そのディスクをシステムディスクとしない場合は、その領域には通常のファイルが格納されるため、その場合はシステムディスクに比べ、使用可能なディスク容量が2つのシステムファイル分増加することになります。

また、この2つのファイルにはシステム属性が付けられ、DIR コマンドや COPY コマンドではアクセスできないようになっています。このため、IO.SYS と MSDOS.SYS の2つのファイルをコピーするには、システムコピー専用のコマンド **SYS** が用意されているのです。SYS コマンドは、この2つのシステムファイルを転送するコマンドですが、このコマンドでシステムをコピーするには、転送先のディスクがフォーマットしたままの空ディスクか、あるいは、データが書き込まれている場合は、コピー元のシステムディスクとまったく同じサイズの IO.SYS と MSDOS.SYS によるシステムが書き込まれている場合に限りますので注意してください。

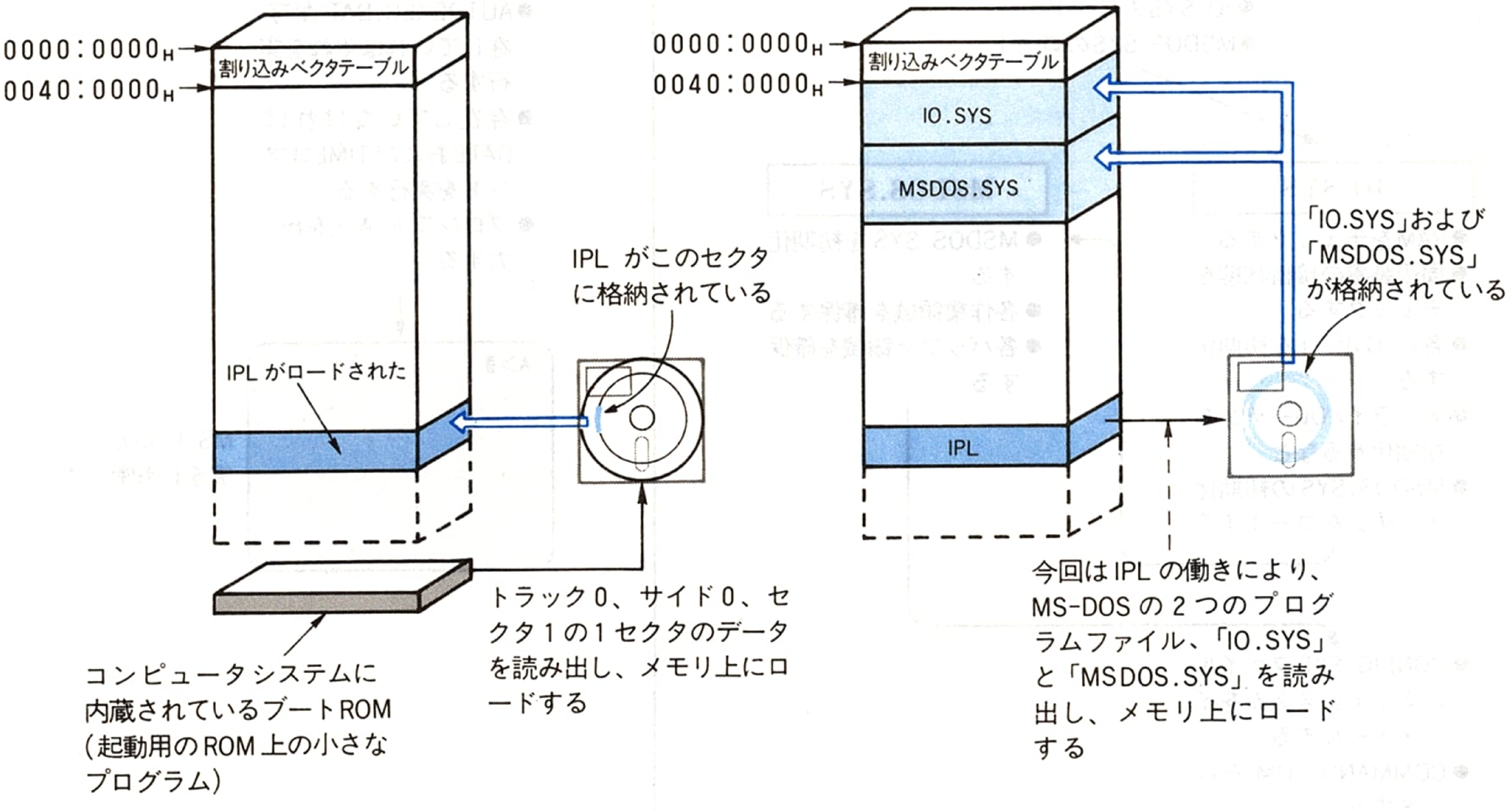

0000:0000H
0040:0000H
割り込みベクタテーブル
IPL がロードされた
IPL がこのセクタに格納されている
トラック0、サイド0、セクタ1の1セクタのデータを読み出し、メモリ上にロードする
コンピュータシステムに内蔵されているブートROM（起動用のROM上の小さなプログラム）
0000:0000H
0040:0000H
割り込みベクタテーブル
IO.SYS
MSDOS.SYS
IPL
「IO.SYS」および「MSDOS.SYS」が格納されている
今回はIPLの働きにより、MS-DOSの2つのプログラムファイル、「IO.SYS」と「MSDOS.SYS」を読み出し、メモリ上にロードする

図 3.9　IPL による IO.SYS および MSDOS.SYS のロード

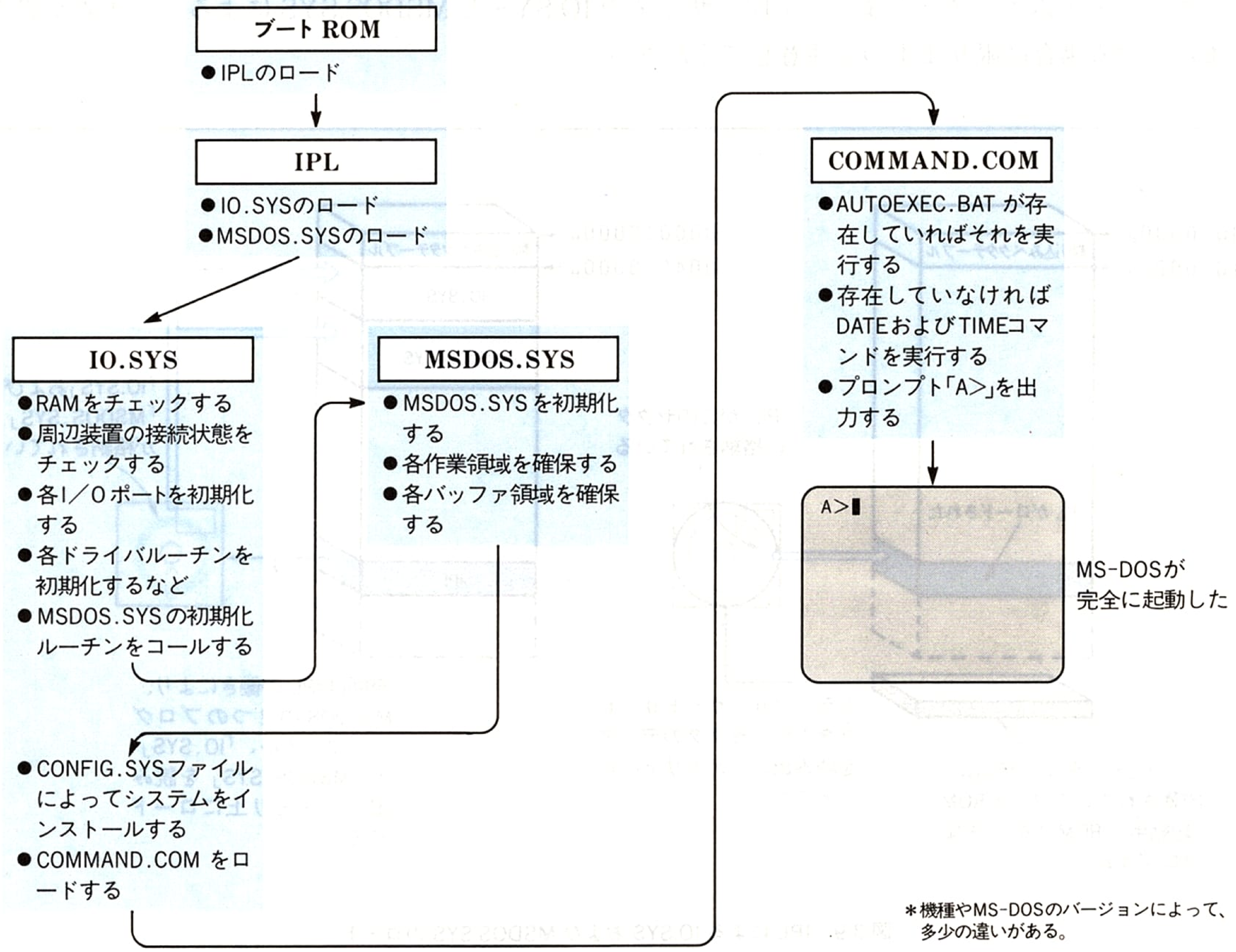
---

* システムディスクかどうかの判別など，多少の付加機能を持たせてある。

3.2.2　ハードウェアとMS-DOSの初期化

IPLによって2つのシステムファイルがメモリ上にロードされると、IPLの役目は終わり、制御の主体はIO.SYSに移ります。IO.SYSは、まず初めに各種の初期化を行います。具体的には、周辺装置の接続状態の確認とその初期化、使用可能なRAM領域のチェック、メモリ上の各種作業領域の確保とその初期化などです。その処理が終わると、MSDOS.SYSの初期化ルーチンをコールし、MS-DOSシステム本体の初期化を行います。

図3.10に、これまでに解説した処理を含めて、MS-DOS起動の開始から終了までの全シーケンスの概要を示しておきましょう。


ブートROM
●IPLのロード
IPL
●IO.SYSのロード
●MSDOS.SYSのロード
IO.SYS
●RAMをチェックする
●周辺装置の接続状態をチェックする
●各I／Oポートを初期化する
●各ドライバルーチンを初期化するなど
●MSDOS.SYSの初期化ルーチンをコールする
MSDOS.SYS
●MSDOS.SYSを初期化する
●各作業領域を確保する
●各バッファ領域を確保する
●CONFIG.SYSファイルによってシステムをインストールする
●COMMAND.COMをロードする
COMMAND.COM
●AUTOEXEC.BATが存在していればそれを実行する
●存在していなければDATEおよびTIMEコマンドを実行する
●プロンプト「A>」を出力する
A>
MS-DOSが完全に起動した

＊機種やMS-DOSのバージョンによって、多少の違いがある。

図3.10　MS-DOS起動の全シーケンス

3.2.3　CONFIG.SYSファイルによるシステムのインストール

MS-DOS本体の初期化が終わると、制御権は再びIO.SYSに戻り、次にCONFIG.SYSファイルによるMS-DOSシステムのインストール(組み込み設定)を行います。システムの初期化に関してぜひ知っておきたいのが、何よりもこのCONFIG.SYSファイルによるシステムのインストールです。

CONFIG.SYSファイルは、普通の(TYPEコマンドで表示可能な)テキストファイルであり、その内容は、

コマンド＝パラメータ

という行を並べたものです。このファイルに目的のコマンドラインを登録しておくことにより、ユーザー各自が、MS-DOSシステムに独自の機能を追加したり、変更したりすることができるのです。MS-DOSのシステム内部を書き換えることなしにその機能を変更できる仕組みはたいへん便利であり、MS-DOSを柔軟性のある使いやすいOSにしています。

これまでの一般的なOSでは、システムの機能を変更したり追加したりするのは、たいへんな苦労を伴う作業でした。たとえば、標準でサポートされているディスクドライブ以外のドライブを、自作のインターフェイスで接続したい場合などは、OS内部のディスクの入出力に関係する部分を書き換えるか、それが不可能であれば、システムとは関係なく、自分のプログラムから直接操作するしかありません。後者の場合は、そのドライブはシステムに認識されないので、システム上のドライブとしては、まったく動作しません。つまり、通常のコマンドやアプリケーションプログラムからは使えないことになります。かといって、システムにそのドライブを認識させるには、前者のようにシステム内部の書き換えが必要であり、MS-DOSのIO.SYSにあたる部分を作り換えることになり、これはたいへんな作業です。

ところが、MS-DOSでは、CONFIG.SYSファイルにコマンドを登録することによって、システムのIO.SYS部に任意の機能を追加したり、その一部を置き換えたりすることができるのです。このCONFIG.SYSの機能により、MS-DOS上の多種多様なアプリケーションプログラムが出現することになったわけで、プログラム開発者にとって、もしこの機能がなかったら、MS-DOSの魅力は半減するといえるほど重要なものなのです。

さて、MS-DOSの起動の話に戻りましょう。MS-DOS本体の初期化が行われたあと、IO.SYSはCONFIG.SYSファイルを捜し、それが存在すると、そのファイルに書かれているコマンドに従って、システムの設定値を変更したり、ユーザーが用意したデバイスドライバ(本章の3.8で解説)をシステムに組み込んだりします(**図3.11**)。

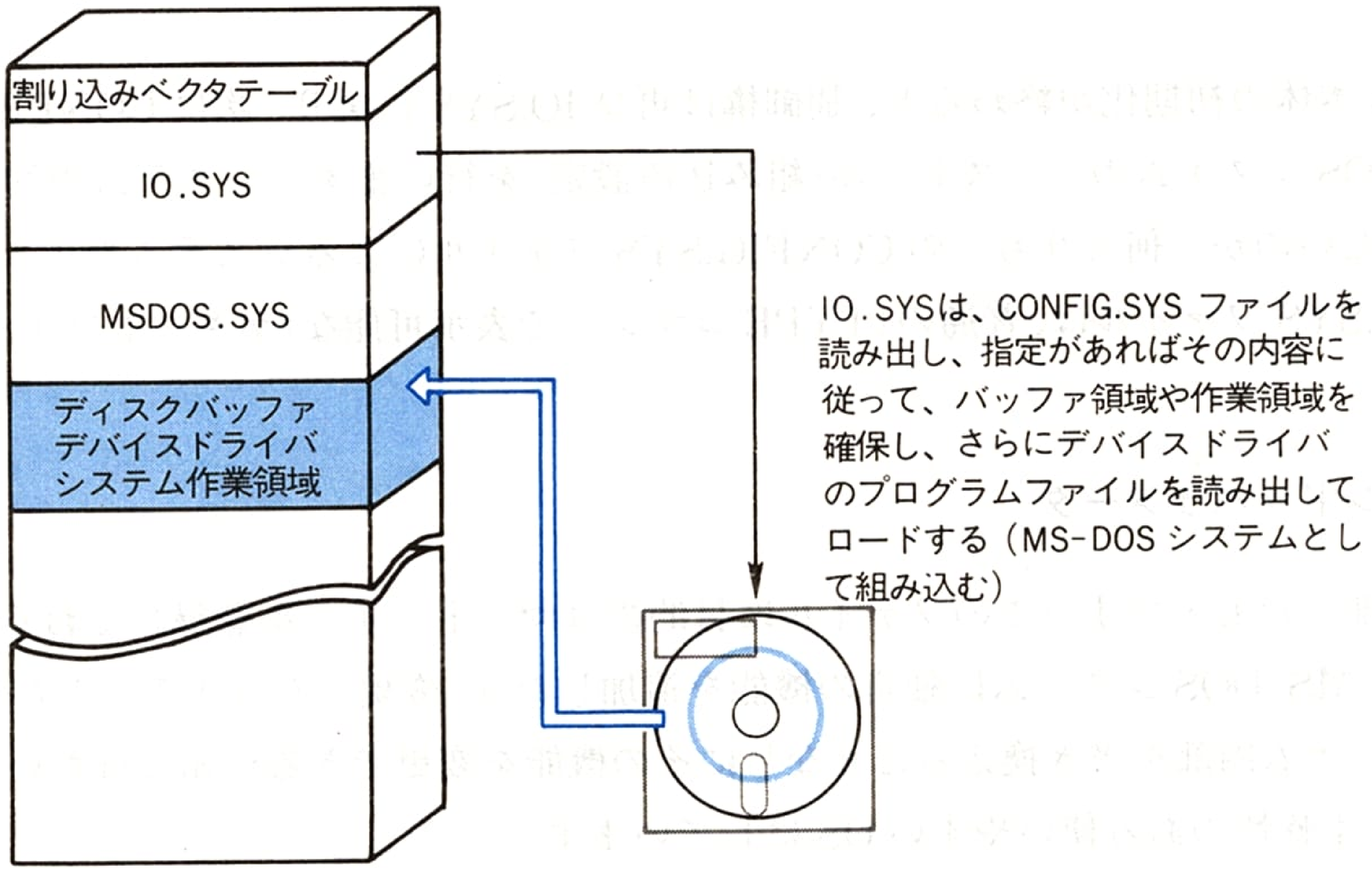

割り込みベクタテーブル
IO.SYS
MSDOS.SYS
ディスクバッファ
デバイスドライバ
システム作業領域
IO.SYSは、CONFIG.SYS ファイルを読み出し、指定があればその内容に従って、バッファ領域や作業領域を確保し、さらにデバイスドライバのプログラムファイルを読み出してロードする（MS-DOS システムとして組み込む）

図 3.11　CONFIG.SYS によるシステムのインストール

■ CONFIG.SYS ファイルに登録できるおもなコマンド

CONFIG.SYS ファイルに登録できるおもなコマンドには次のようなものがあります。それぞれについて簡単に解説していきましょう。

FILES＝ファイル数

2 章で解説しましたが、私たちはファイルハンドルを使うことにより、ファイルを簡単にアクセスすることができます。しかしそのためには、システム内に FCB 用の領域や、ファイルハンドルと FCB との対応を示すテーブルなどの領域が必要です。FILES コマンドの「ファイル数」というのは、ユーザープログラムが動作するには、一度にいくつかのファイルをオープンしなければならないので、それに対応するために、システムに対してその上限を指定しておくための数です。なお FILES コマンドによる数の指定を省略した場合は、デフォルト（初期値）として 8（MS-DOS バージョン 2.*x* では 5）が与えられていますので、ユーザーがアクセス可能なファイル数は、8 個（同、5 個）ということになります。これ以上のファイルを、一度にオープンしてアクセスする必要があるユーザープログラム（たとえばデータベースソフトなど）を実行する場合は、ファイル数を多く指定することができます。ただし、1 つのプロセス（プログラムのことだと思えばよい。後述）では 20 個までのファイルをオープンすることができますがファイルハンドルの最初の 5 つは、システムが使用するために予約済みなので、ユー

ザーが同時にオープンできるファイルの数は最大15個となります。FILESの指定は通常15〜16ぐらいが適当でしょう。このファイル数を必要な数だけ設定していないと、ファイルをオープンできないというエラーメッセージを出力して実行がストップします。ディスク上にファイルが存在しているのにオープンできない場合は、たいていこのFILESの数を指定していないか、あるいはその数が不足しているかのどちらかです。

BUFFERS＝バッファ数

これは、ディスク・バッファリングのためのバッファ数のことで、デフォルトは、メインメモリの容量によって、5、10、20と、3段階に変化します（MS-DOSバージョン2.*x*では2に固定。2章の2.2.3参照）。2章で解説しましたが、ディスクが交換されたかどうかを検出できるディスクドライブを接続しているシステムならば、このディスクバッファの数を多くとることによってディスクアクセスの速度が大幅に向上します。ディスクの交換が検出できないシステムでは、この数をいくら大きくしても効果はありません。また、このバッファ数をあまり大きくしすぎても、実際に登録されているファイルの数とのかねあいで、目的のセクタがバッファに取り込まれているかどうかを調べるための内部処理の時間が無視できなくなり、かえって逆効果になりかねません。普通は15〜20くらい、またファイルの数が極端に多い場合（数百、数千）は40〜50などがよいでしょう。

BREAK＝ON　（またはOFF）

これは、内蔵コマンドのBREAKコマンドと同じです。これがONの場合は、すべてのシステムコールの実行中に、Ctrl-Cの入力チェックが行われます。これはデフォルトではOFFになっています。

DEVICE＝デバイスドライバのファイル名［オプション］

前節でも触れていますが、MS-DOSシステムに各種周辺装置（デバイス）のユーザー独自の基本操作プログラム（**デバイスドライバ**）を組み込むためのコマンドがこれです。このDEVICEコマンドこそ、MS-DOSの優れた柔軟性を発揮させるためのコマンドです。ユーザー独自のデバイスをシステムに組み込むには、そのデバイスドライバを作成して、そのプログラムファイル名をこのコマンドに書いておくだけでよいのです。デバイスドライバとは、ディスクドライブやプリンタやマウスなど各種の周辺装置を操作するプログラムのことで、詳しくは本章の3.8節で解説します。また、7章ではデバイスドライバのプログラムの作成実習を行います。

なお、3.8.9で解説するように、ADDDRVコマンド（MS-DOSバージョン3.*x*より付属）によって、システム起動後にデバイスドライバを組み込むこともできます。

SHELL＝コマンド・プロセッサのパス名［オプション］

本章の3.1で述べましたが、ユーザーの入力したコマンドラインを解釈して実行する部分がCOMMAND.COMであり、**コマンド・プロセッサ**と呼ばれているものです。これはMS-DOSのユーザーインターフェイスの部分であり、MS-DOSの顔といってもよいでしょう。COMMAND.COMには、MS-DOSの内蔵コマンドの各プログラムが含まれているわけですが、これらのコマンドの機能を変更したものや、自分で作った別のコマンド・プロセッサを、SHELLコマンドを使って入れ換えたりすることができます。つまり、ユーザーが独自に作ったコマンド・プロセッサのファイル名を、SHELLコマンドで指定するだけで、標準のCOMMAND.COMと入れ換えることができ、MS-DOSを自分独自の環境にすることができるのです。そのほかにもこのSHELLコマンドには、パス名を含めたCOMMAND.COMを指定することによって、ルートディレクトリ以外に置く機能があります。

以上解説したCONFIG.SYSに関する主要コマンドの実例を図3.12に示します。

```
A>TYPE CONFIG.SYS↵ ……………CONFIG.SYSファイルの内容を表示する
FILES=15 ……………一度にオープン可能なファイル数を15個に設定する
BUFFERS=10 ……………ディスク・バッファリングのためのバッファ数を10に設定する
DEVICE=MOUSE.SYS ……………マウスのデバイスドライバをシステムに組み込む
SHELL=A:¥BIN¥COMMAND.COM A:¥BIN /P ……………MS-DOSの起動時に、
A>             COMMAND.COMをAドライブのディレクトリ「¥BIN」から読み込
               む。また、再ロード時も同じ。いずれも/Pオプション付き(3.7.1で解説)
```

CONFIG.SYSフィイルの設定の例

図3.12　CONFIG.SYSファイルの例

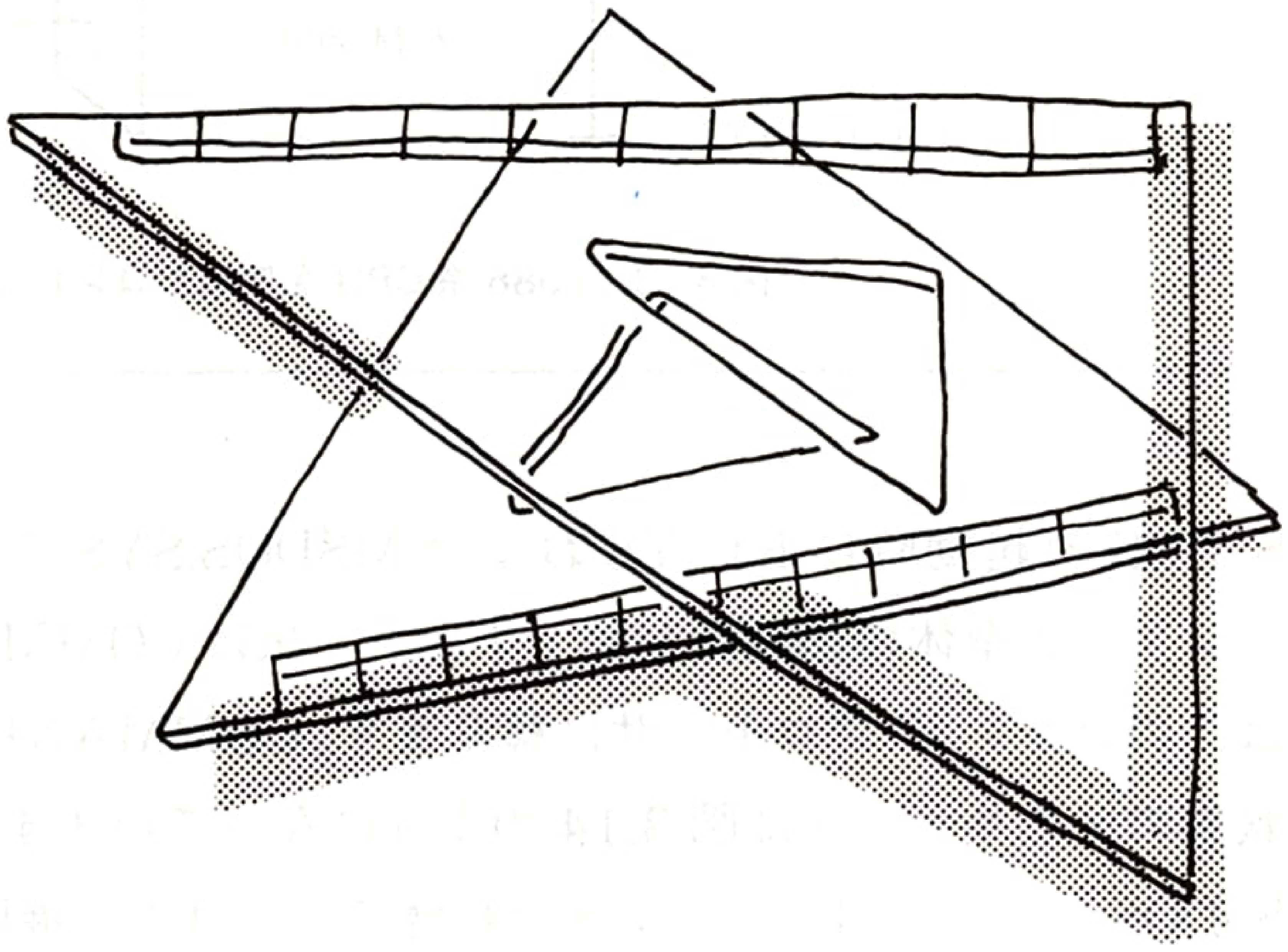
3.2.4　COMMAND.COMのロード

では、MS-DOSの起動の話に戻りましょう。

CONFIG.SYSファイルに記述されたコマンドにより、ディスクバッファの確保や、デバイスドライバの組み込みなどが行われ、システムのインストールが終了すると、IO.SYSは次にコマンド・プロセッサをロードします。CONFIG.SYSに、前述のSHELLコマンドの指定がなければ、Aドライブのルートディレクトリから COMMAND.COMを捜してロードします。なお、その際COMMAND.COMは/Pオプション付きでロードされます。この/Pオプションについては、本章の3.7.1の「プロセス管理」のところで解説しますが、COMMAND.COMをシステムに常駐させるというオプションコマンドです（図3.12参照）。

3.2.5　COMMAND.COMの起動

IO.SYSによるCOMMAND.COMのロードが終了すると、COMMAND.COMが起動され、プログラムの制御権はCOMMAND.COMに移ります。COMMAND.COMが実行されると、まず、ルートディレクトリ上のオートスタート・バッチファイル(AUTOEXEC.BAT)が捜され、それが存在していれば、その内容を自動的に実行する処理に移ります。もし存在していなければ、DATEコマンドおよびTIMEコマンドを順に実行し、この2つのコマンドの入力要求に答え終わると、プロンプト「A>」が出力され、MS-DOSは完全な起動状態となります(図3.10参照)。



3.3　MS-DOSの基本構成

本節では、MS-DOSが完全に起動して、システムが動作状態のとき、MS-DOSの各システムはメモリ上にどのように展開しているのか、その構成などを見ていきましょう。

MS-DOSマシンのメモリ構成は、8086系CPUのハードウェア的な制約から、通常は下位アドレスにはRAM、上位アドレスにはROMを配置します。MS-DOSは、もともとこのようなシステムを仮定して作られているのです。8086系CPUでは、割り込みベクタテーブルがメモリの最下位に置かれ、リセット時にはメモリの上位アドレスから実行が開始されるため、このようなRAM、ROM配置になるわけです(図3.13参照)。

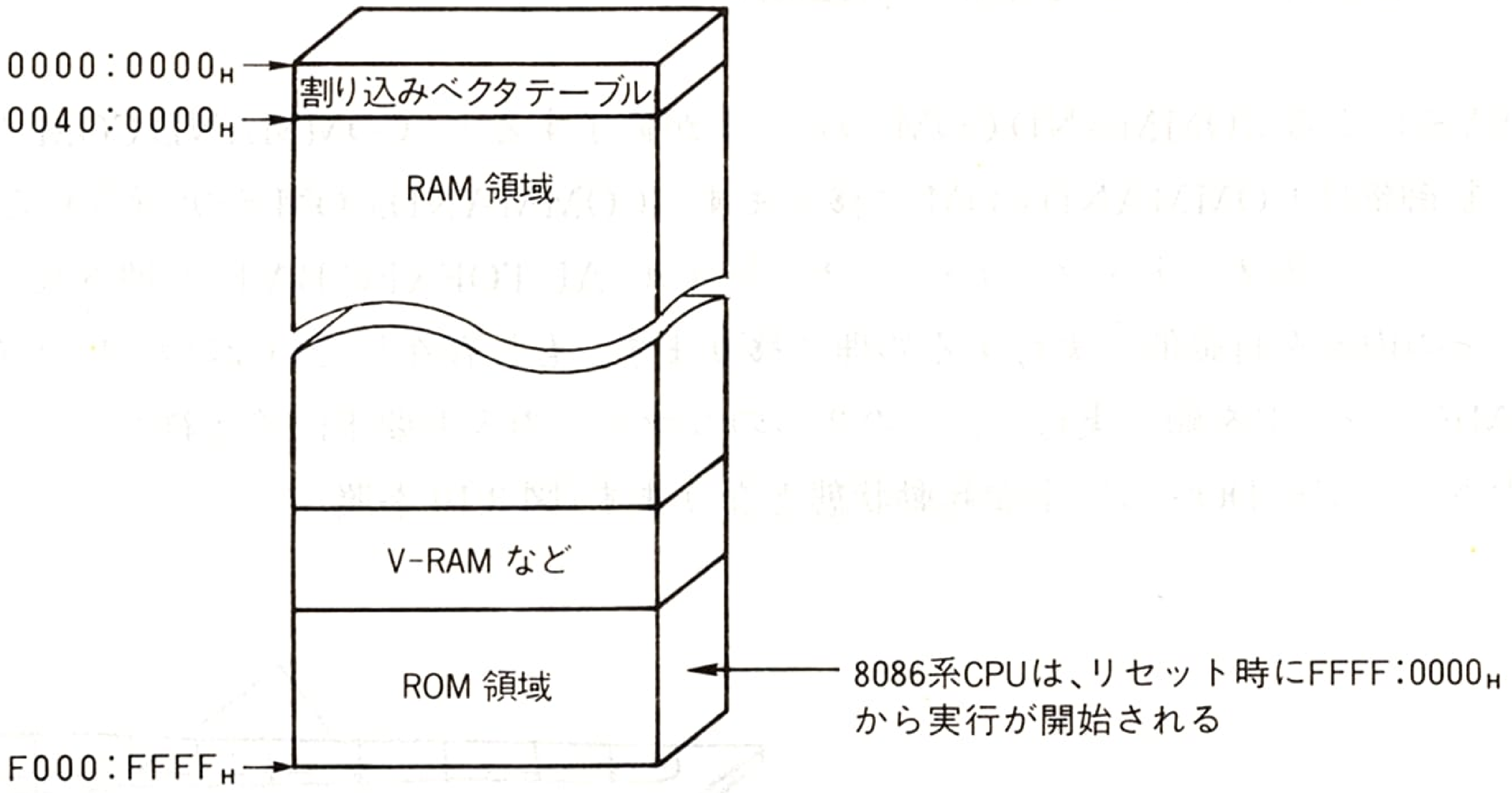
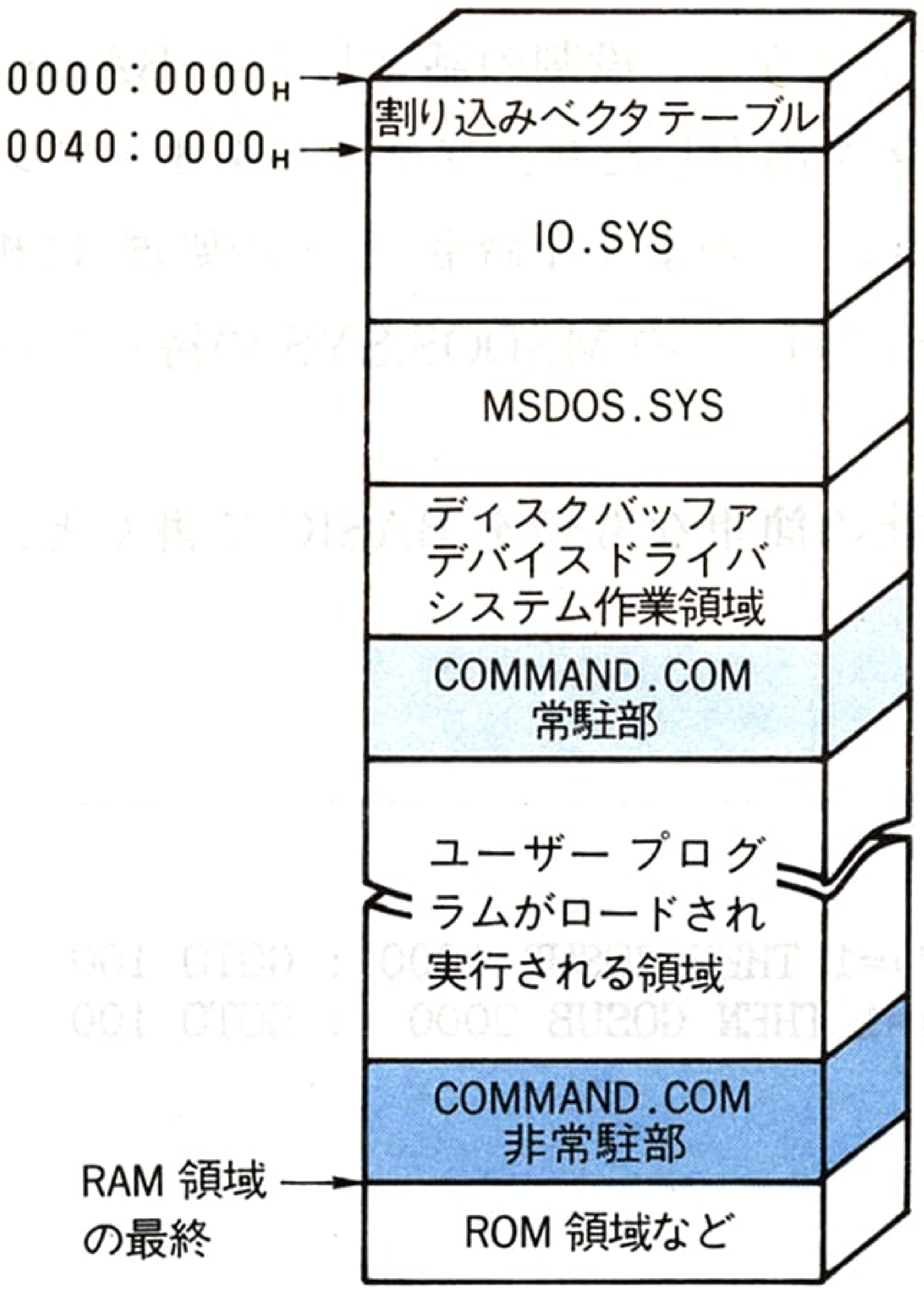

0000：0000H
0040：0000H
割り込みベクタテーブル
RAM 領域
V-RAM など
ROM 領域
F000：FFFFH
8086系CPUは、リセット時にFFFF：0000H
から実行が開始される

図 3.13　8086 系 CPU を使ったコンピュータの基本メモリ構成

MS-DOS の起動時に IO.SYS および MSDOS.SYS の 2 つのシステムがロードされ、IO.SYS によってシステム全体の初期化が完了すると、次に COMMAND.COM がロードされ、プログラムの制御権はこのコマンド・プロセッサに移ります。COMMAND.COM が実行され、MS-DOS が完全に起動した状態のメモリマップは図 3.14 のようになっています。

MS-DOS システムは、このような状態でメモリ上に展開され動作しています。このとき、任意のコマンドを入力することにより、それが MS-DOS の内蔵コマンドであれば COMMAND.COM 内の該当プログラムが実行され、外部コマンドやユーザープログラムであれば、ディスクからそのプログラムがユーザーRAM 領域にロードされて実行されます。

ところで、COMMAND.COM はたいへん多くの機能を持つ優れたコマンド・プロセッサですが、その機能の大部分は、ユーザープログラムなどの外部プログラムを実行している間は必要ありません。そこで、外部プログラムを実行する際に、COMMAND.COM はそのほんの一部を残して今まで自分が占有していたメモリ領域を解放し、そこへ外部プログラムをロードして実行します。このため、MS-DOS の起動時にロードされた COMMAND.COM は、図 3.14 のように 2 つの部分に分かれているのです。



0000:0000H
0040:0000H
割り込みベクタテーブル
IO.SYS
MSDOS.SYS
ディスクバッファ
デバイスドライバ
システム作業領域
COMMAND.COM
常駐部
ユーザー プログラムがロードされ実行される領域
COMMAND.COM
非常駐部
RAM 領域の最終
ROM 領域など

図 3.14　MS-DOS が完全に起動した状態のメモリマップ

3.4　内蔵コマンド実行時の各部の働き

本節では、MS-DOS の内蔵コマンドの実行時に、IO.SYS、MSDOS.SYS、COMMAND.COM の各部が、どのような働きをするのかを見ていきましょう。なお、ここでは、それぞれの解説が内蔵コマンドの実行という面から述べられていますので注意してください。

3.4.1　COMMAND.COM

COMMAND.COM は、ユーザーと MS-DOS との接点の部分であり、ユーザーが入力したコマンドを受け取り、それを解釈し、その要求が内蔵コマンドの実行であれば、COMMAND.COM 内部に用意されている該当するプログラムを実行に移します。それぞれのコマンドを実行するためのプログラムは、普通のユーザープログラム(外部プログラム)と何ら変わることはありません。1章で作成した CHMOD プログラムと同じように作られていると思えばよいでしょう。ということは、それらのプログラムには、当然 MSDOS.SYS の機能(つまりシステムコール)も使われているはずです。

■ COMMAND.COMはBASICのプログラム？

COMMAND.COMは、ちょうどBASICで書いたプログラムのようなものと考えることができます(実際にBASICで書かれているわけではなく、役割の話として)。BASICには、キーボードから文字列を入力したり、ディスプレイに文字を出力したり、ファイルをオープンしたり、それを読み出したりする命令があります。MS-DOSでは、このような命令(とその処理)に相当するものは、MSDOS.SYS内にあります。そのためMS-DOSでは、このMSDOS.SYSの持っている命令を利用してプログラムが書けるのです。

たとえば、COMMAND.COMの機能の簡単なものをBASICで書くと、**図3.15**のようになるでしょう。

```
100 INPUT "A>",COMMAND$
200 IF INSTR(COMMAND$,"TYPE ")=1 THEN GOSUB 1000 : GOTO 100
210 IF INSTR(COMMAND$,"DIR ")=1 THEN GOSUB 2000  : GOTO 100
    ⋮

520 GOTO 100
    ⋮

1000 FILE$=RIGHT$(COMMAND$,LEN(COMMAND$)-LEN("TYPE "))
1010 OPEN FILE$ FOR INPUT AS #1
1020 IF EOF(1) THEN GOTO 1060
1030    INPUT #1,C$
1040    PRINT C$
1050 GOTO 1020
1060 CLOSE #1
1070 RETURN
    ⋮

2000 FILES
2010 RETURN
```

［このリストについての解説は行いませんので、各自で考えてみてください］

図3.15　COMMAND.COMの機能の一部をBASICで書いてみる

このBASICプログラムでのINPUT文やOPEN文は、実際のCOMMAND.COMでは、MSDOS.SYS内のそれに対応する命令を呼び出すことによって実現しています(これが4章で解説するシステムコールである)。

ここでの解説で、COMMAND.COMの役割とMSDOS.SYSとの役割の分担について厳密に解説しようとすると、たいへんめんどうなことになってしまします。たとえば、TYPEコマンドについての説明では、こんなことになるでしょう。

> COMMAND.COM内のTYPEコマンドを実行するプログラムは(すでにユーザーの要求がTYPEコマンドであることは解釈済みとして)、指定されたファイルをMSDOS.SYSのファイルオープンのシステムコールを使ってオープンし、MSDOS.SYSのファイル読み出しのシステムコールを使ってファイルを読み出し、MSDOS.SYSのコンソールへ文字を出力するシステムコールを使って読み込んだ内容をコンソールに表示し、これをファイル読み出しのシステムコールがEOF(End Of File：ファイルの終わり)を検出するまで繰り返し、これが検出されると、MSDOS.SYSのファイルクローズのシステムコールを使ってファイルをクローズして、このTYPEコマンドを終わる。

この説明を、COMMAND.COMとMSDOS.SYSとの役割を区別せずに、普通に表現すると、

> COMMAND.COM内のTYPEコマンドを実行するプログラムは(すでにユーザーの要求がTYPEコマンドであることは解釈済みとして)、指定されたファイルをオープンし、ファイルを読み出してそれをコンソールに出力し、これをEOFが検出されるまで繰り返し、検出されたらファイルをクローズしてTYPEコマンドを終わる。

となります。後者の方が、全体の流れはよくわかると思いますが、それぞれの処理のどの部分をCOMMAND.COMが実行し、どの部分をMSDOS.SYSが実行するのか、区別がつきません。これは、BASICのプログラムでも同じことがいえます。

たとえば、OPEN文によるプログラムを実行する際、実際にはBASICインタープリタにファイルをオープンしろと命令しているだけなのに、私たちは「ファイルをオープンする」といい、あなたの書いたOPEN文のプログラムそのものが直接ファイルをオープンするようないい方をします。実際はそうではなく、ファイルをオープンするのは、「OPEN」という文字列をBASICインタープリタが解釈して、その結果、BASIC内のファイルオープンのマシン語プログラムに実行させるのです。

このようなことは、どんなプログラムにもいえることで、実際はサブルーチンを呼び出して行う処理なのに、直接その処理を行っているように表現するのが普通です。本章の解説のように、役割の分担について話している場合には、これをはっきり区別しなければ中途半端な解説になる恐れもありますが、これを1つひとつ区別すると、さきの例のように煩雑な文章となり、かえって本筋がわからなくなるでしょう。したがって、これからの解説も「普通」に表現することにします。

MS-DOSに限らず、OSの役割がよくわからないとか、ファイルを扱うプログラムをマシン語で書けないという人たちは、OSがどこまでのことをしてくれるのか、その範囲を理解していない場合が多いようです。これは、プログラム作りの苦労を重ねるうちに次第にわかってくるものであり、逆にそれがわかってくれば、OSのことがわかってきたことにもなります。本当にMS-DOSを理解したいと思うならば、マシン語レベルでプログラムを作ってみることです。まずは1章、2章、4章などにある実習プログラムを作成することから始めればよいでしょう。

さて話がジャンプしましたが、COMMAND.COMはユーザーのコマンドを解釈して実行に移すという点では、MS-DOSのシステムの一部ではありますが、プログラムの構造上は、一般のユーザープログラムなどとなんら変わりはありません。必要であれば、あなたが自身が独自のコマンド・プロセッサを作ることも可能なのです。COMMAND.COMの働きは図3.16を参照してください。

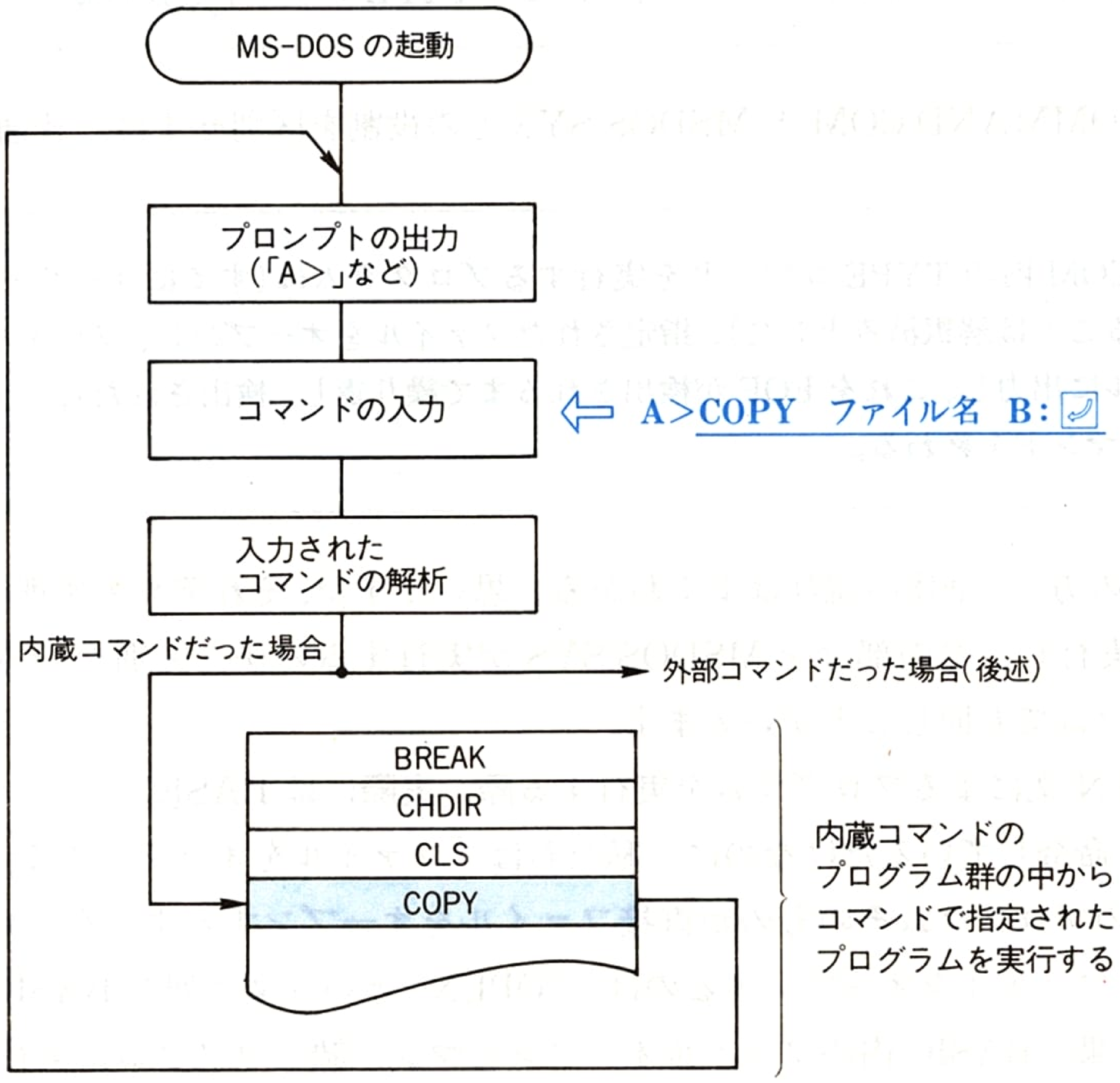

MS-DOSの起動
プロンプトの出力
(「A>」など)
コマンドの入力
A>COPY　ファイル名　B:⏎
入力された
コマンドの解析
内蔵コマンドだった場合
外部コマンドだった場合(後述)
BREAK
CHDIR
CLS
COPY
内蔵コマンドの
プログラム群の中から
コマンドで指定された
プログラムを実行する

図3.16　COMMAND.COMの役割(内蔵コマンドの場合)

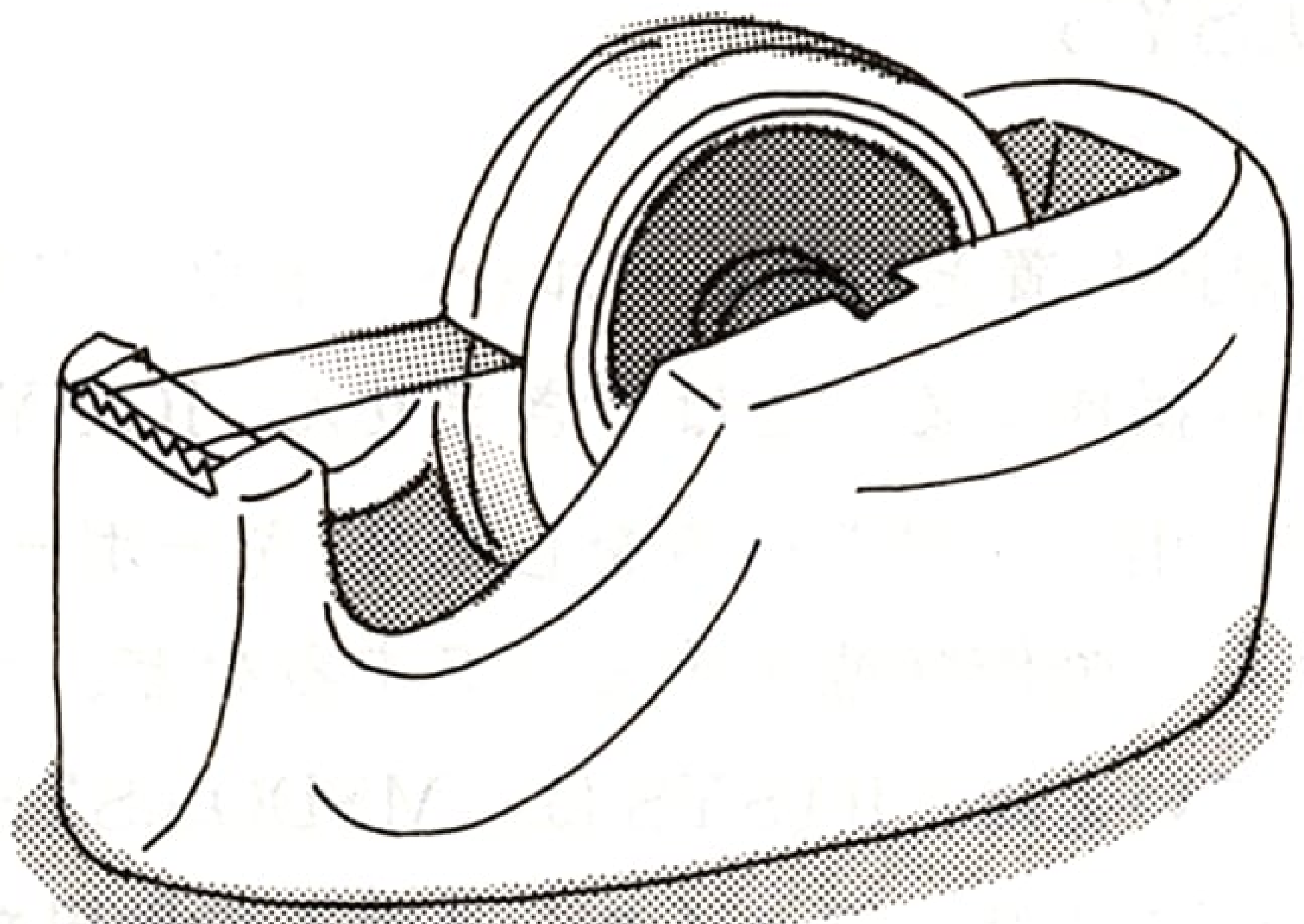
3.4.2　MSDOS.SYS

MSDOS.SYSの役割は主にファイル管理であり、そのほかにメモリ管理、周辺装置（デバイス）の管理などを行います。メモリ管理を除いて、ほとんどの仕事は、外部プログラムやIO.SYSとの間の処理であり、IO.SYSが行う単純な入出力を利用して、複雑なファイルシステムなどを実現する部分です。

MSDOS.SYSが行うことは、キーボードから文字列を入力する、ファイルをオープンするなどの、ある程度のまとまりを持った「仕事」で、その1つひとつは、BASICの命令や、C言語の標準ライブラリ関数（6章参照）のようなものに相当します。

MSDOS.SYSは、COMMAND.COMや外部プログラムからの、ファイルのオープンや、ディスプレイへの1文字出力などの要求が発生するたびに、IO.SYSのそれぞれのルーチンを呼び出すことによって、ディスクの必要なセクタをアクセスしたり、ディスプレイへ文字を出力したりして要求を実現します。もう少し具体的に解説すると、たとえば外部プログラムからファイルをオープンする要求があれば、MSDOS.SYSは、IO.SYS内のセクタの読み出し／書き込みルーチンを呼び出すことによって、ディレクトリが格納されているセクタを読み出し、そのデータから目的のファイル名を検索し、見つかればFCBなどのテーブル類を作成して、オープン成否の結果を外部プログラムに返す処理を行うのです。ファイルの読み出しの要求であれば、MSDOS.SYSはオープンしたときに作成したテーブルを参照して、目的のファイルが何番目のセクタにあるかなどを計算し、IO.SYS内のセクタ読み出し／書き込みのルーチンを呼び出し、そのセクタを読み出して、ユーザーが指定したバイト数分のデータを、指定したバッファに転送します。

ユーザープログラムを作成する際に、MSDOS.SYSがどこまでやってくれ、自分がどこまでのプログラムを書かなければならないのかは、おいおいわかってくるでしょう。MSDOS.SYSは、各種の機能のかなりの部分を内部でやってくれるので、ユーザーは目的の機能に対していくつかのパラメータを渡すだけで、簡単にその処理を行うことができます（詳しくは4章で解説する）（図3.17参照）。



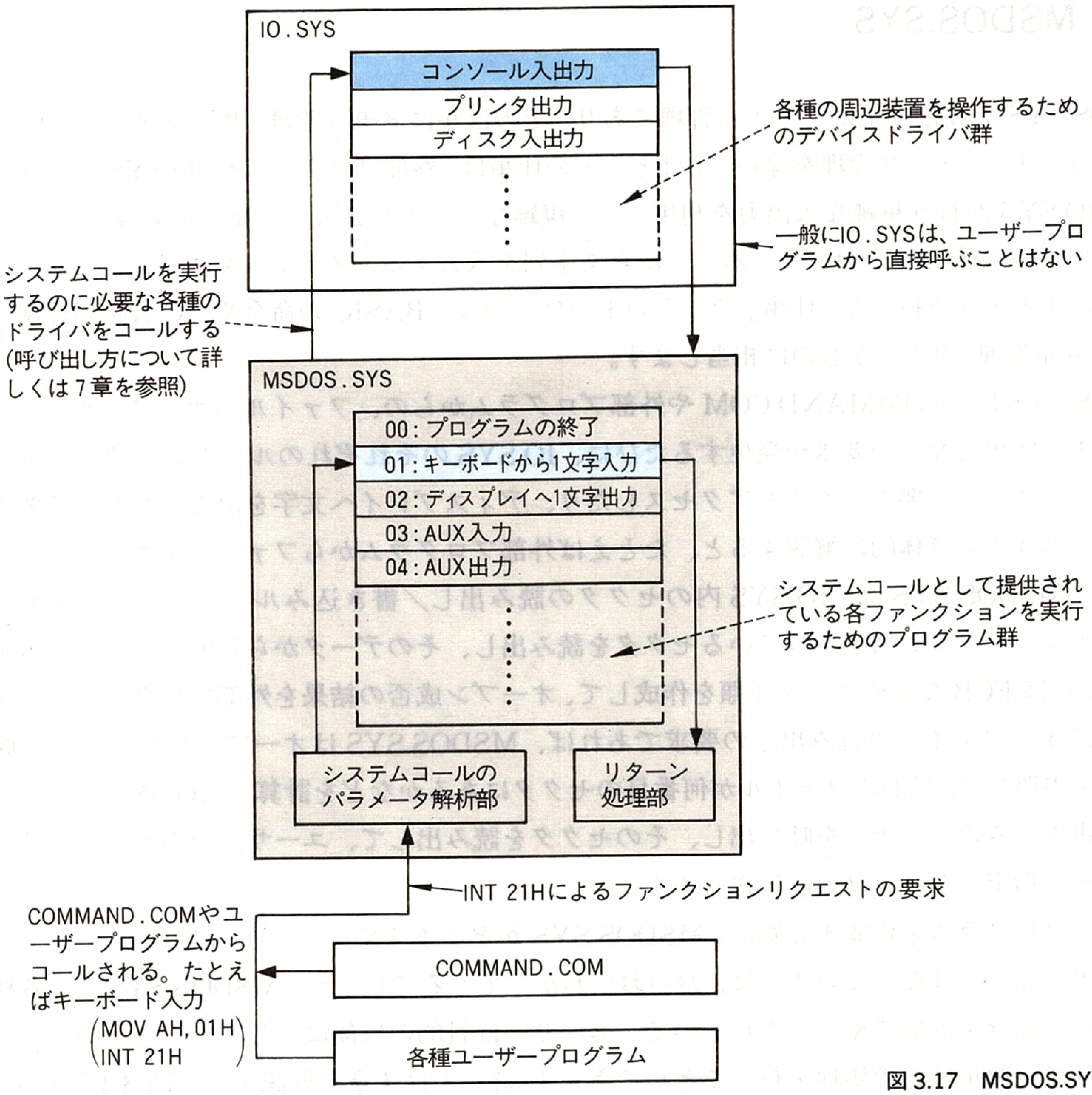

IO.SYS
コンソール入出力
プリンタ出力
ディスク入出力
各種の周辺装置を操作するためのデバイスドライバ群
一般にIO.SYSは、ユーザープログラムから直接呼ぶことはない
システムコールを実行するのに必要な各種のドライバをコールする(呼び出し方について詳しくは7章を参照)
MSDOS.SYS
00：プログラムの終了
01：キーボードから1文字入力
02：ディスプレイへ1文字出力
03：AUX入力
04：AUX出力
システムコールとして提供されている各ファンクションを実行するためのプログラム群
システムコールのパラメータ解析部
リターン処理部
INT 21Hによるファンクションリクエストの要求
COMMAND.COMやユーザープログラムからコールされる。たとえばキーボード入力
MOV AH, 01H
INT 21H
COMMAND.COM
各種ユーザープログラム

図 3.17　MSDOS.SYS の役割

3.4.3　IO.SYS

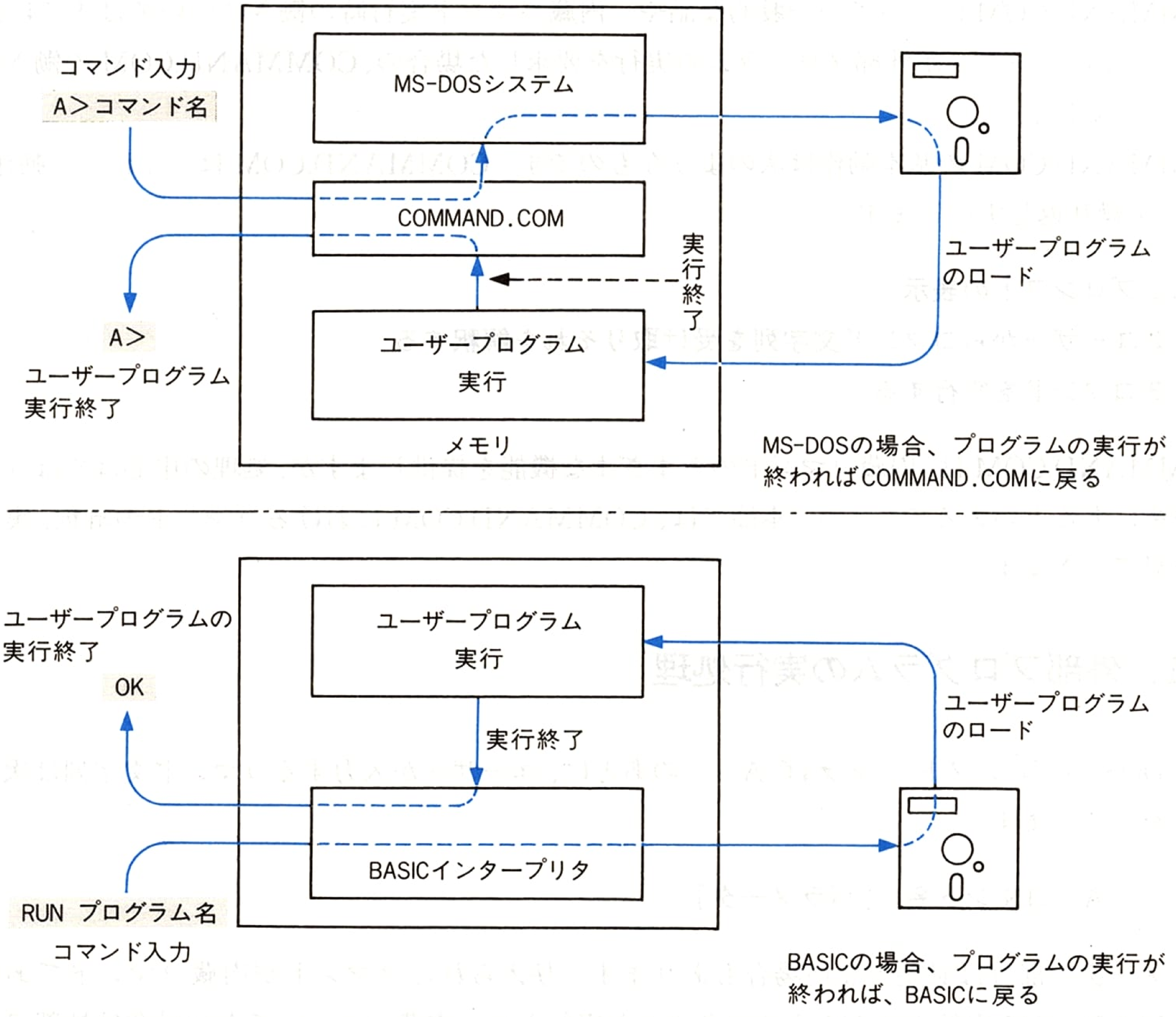
IO.SYS は周辺装置との低レベル(ハードウェアを直接操作する)の入出力を行いますが、外部プログラムから直接操作することはできません。IO.SYS は MSDOS.SYS によって呼び出され、ディスクのセクタの読み出し／書き込みをしたり、キーボードやディスプレイとのデータの入出力を行ったり、あるいはそれらの動作状態をチェックするなど、ハードウェアを直接操作しています。ディスクのアクセスについていえば、IO.SYS は、MSDOS.SYS から命令された指定セクタを読み出したり書き込んだりするだけであり、ファイルシステムの世界などは、その存在すら関知していません(図 3.17)。

3.5 外部プログラム実行時の各部の働き

ユーザーが入力したコマンドを COMMAND.COM が解釈した結果、それが内蔵コマンドではなかった場合、COMMAND.COM は外部プログラムの処理に移ります。要求されたプログラムファイルをディスクから捜し、見つかればそのプログラムをロードし実行するのです。本章の 3.4 節で示した BASIC で書いた COMMAND.COM では、この処理はさしずめ、

```
500  RUN  COM$
```

とでもするのでしょう。MSDOS.SYS 内の機能にも、この BASIC の RUN コマンドのようなものが



コマンド入力
A>コマンド名
MS-DOSシステム
COMMAND.COM
実行終了
ユーザープログラムのロード
A>
ユーザープログラム実行終了
ユーザープログラム実行
メモリ
MS-DOSの場合、プログラムの実行が終わればCOMMAND.COMに戻る
ユーザープログラムの実行終了
OK
ユーザープログラム実行
ユーザープログラムのロード
実行終了
BASICインタープリタ
RUN プログラム名
コマンド入力
BASICの場合、プログラムの実行が終われば、BASICに戻る

図 3.18　MS-DOS と BASIC でのユーザープログラム実行の違い

あるわけですが、RUNコマンドと違う点は、MS-DOSでは外部プログラムが終了すると、MSDOS.SYSに戻るのではなく、COMMAND.COMに戻ることです。BASICでは、COMMAND.COMに相当する部分が独立していないため、BASICのユーザープログラムはBASICが実行し、終了するとBASICに戻ります(図3.18)。

COMMAND.COMは、あくまでユーザーとMS-DOSとの間のインターフェイスをとるもので、外部プログラムの実行中には、ほとんど用はありません。外部プログラムを実行するに至るまでの、お膳立てをするものと考えればよいでしょう。

3.6 COMMAND.COMの働き

COMMAND.COMについての一般的な話や、内蔵コマンド実行時の働きについてはすでに述べました。ここではユーザーが外部プログラムの実行を要求した場合の、COMMAND.COMの働きについてみていきましょう。

COMMAND.COMの基本動作は次のようなものです。COMMAND.COMは、これらの動作(①→②→③)を繰り返し実行します。

①プロンプトの表示
②ユーザーからコマンド文字列を受け取りそれを解釈する
③コマンドを実行する

COMMAND.COMは、内蔵コマンドのさまざまな機能を提供しますが、処理の中心はやはりコマンドの解釈にあるといえるでしょう。本節では、COMMAND.COMにおけるコマンドの解釈、実行の仕組みを見ていきます。

3.6.1 外部プログラムの実行処理

MS-DOSのプロンプト(たとえば「A>」)のあとに、ユーザーが入力するコマンド文字列は次のような形式をしています。

```
A>コマンド名　[パラメータ]
```

パラメータの部分は必要のない場合もあります。与えられたコマンドが内蔵コマンドであれば、COMMAND.COM内部の該当するプログラムが実行され、内蔵コマンドでない場合は外部プログラムの処理に移って、ディスク上のディレクトリを検索してそのプログラムファイルを捜します。この

ときの検索は、環境変数 PATH に設定された順番で行います。

具体的には、まずカレントディレクトリ内に存在する、目的のプログラムファイル名の、そのファイルタイプが「.COM」「.EXE」あるいは「.BAT」であるものがこの順に検索され、もしここで該当するプログラムファイルがなければ、環境変数 PATH で設定されたディレクトリの順番に、さきほど

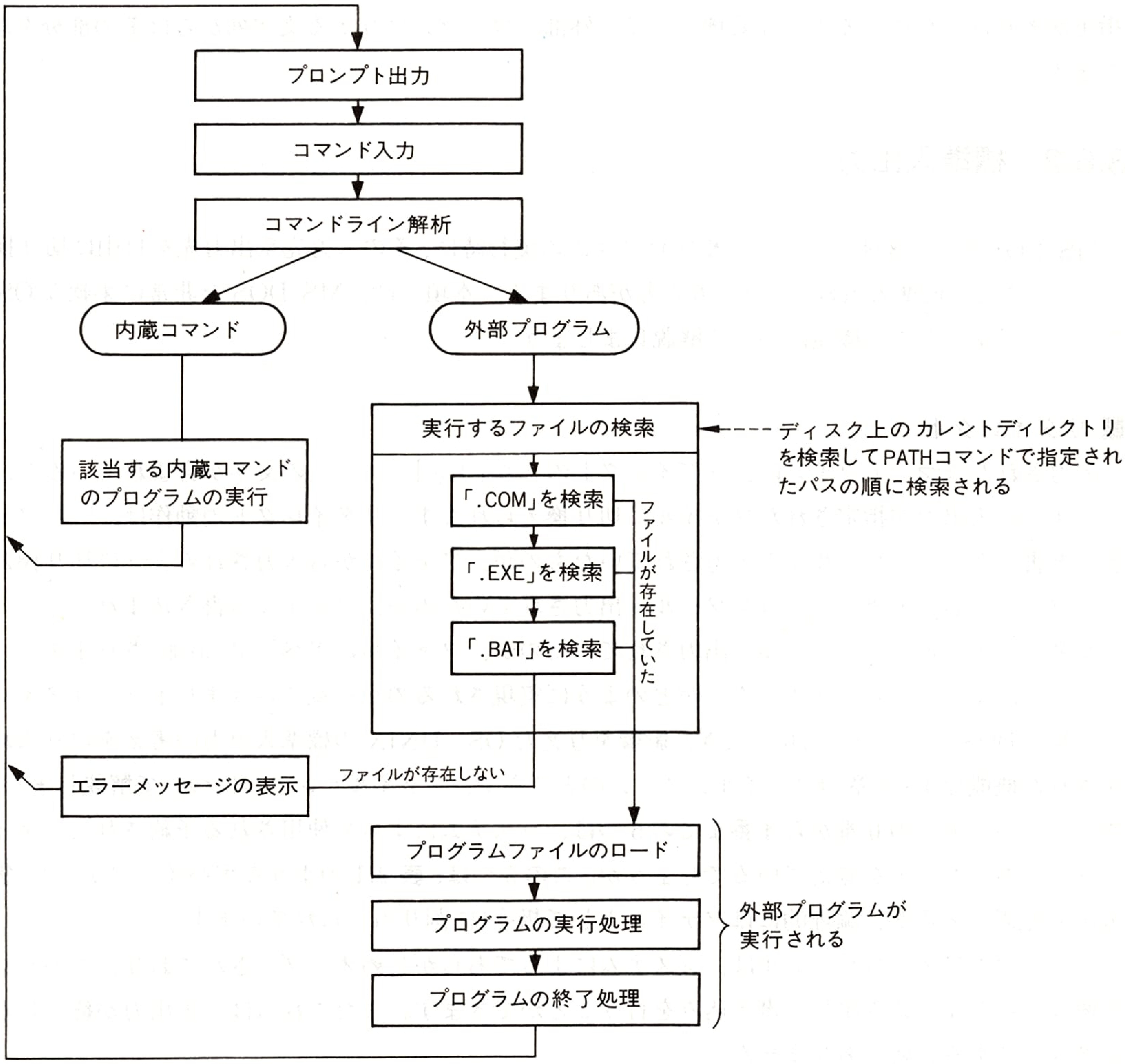

プロンプト出力
コマンド入力
コマンドライン解析
内蔵コマンド
外部プログラム
該当する内蔵コマンドのプログラムの実行
実行するファイルの検索
ディスク上のカレントディレクトリを検索してPATHコマンドで指定されたパスの順に検索される
「.COM」を検索
「.EXE」を検索
「.BAT」を検索
ファイルが存在していた
エラーメッセージの表示
ファイルが存在しない
プログラムファイルのロード
プログラムの実行処理
プログラムの終了処理
外部プログラムが実行される

図 3.19　COMMAND.COM により外部プログラムが実行される過程

と同じファイルタイプの順に検索します。目的のファイルが見つかると、そのプログラムファイルをロードして実行に移します（**図3.19**参照）。

「目的のプログラムがロードされ実行される」と簡単にいっていますが、外部プログラムの実行は、ロードしたプログラムを実行するためのMSDOS.SYS内の処理によって行われます。この処理はのちほど解説しますが、外部プログラムは、COMMAND.COMというプログラム上の**子プロセス**として実行されるのです。実行する外部プログラムには、入力されたコマンド文字列からコマンド名を取り除いた部分と、環境文字列が与えられます。このとき、コマンド文字列に、リダイレクトやパイプの指定があれば、後述するような処理をして、外部プログラムに与える文字列からはその部分を取り除きます。

3.6.2　標準入出力

MS-DOSでは、各種のコマンドやプログラムの実行時に、その入力先や出力先を自由に切り換えることができる「標準入出力」という考え方があります。本項では、MS-DOSを非常に柔軟なOSとしている注目すべきこの機能について解説しましょう。

■ リダイレクト

入力されたコマンドラインに、リダイレクトの「<」「>」「>>」の文字が含まれていると、コンソールへの入出力が指定されたファイルに切り換えられます。リダイレクトの動作は、「< ファイル名」と書くと、コンソールから入力されていたものが、ファイルから入力されるように切り換わり、「> ファイル名」と書くと、コンソールへ出力されていたのが、ファイルへ書き込まれ、「>> ファイル名」と書くと、コンソールへ出力されていたのが、ファイルにアペンド（追加）されます。

ここではまず、このリダイレクトがどのように実現されるのかを見ていきましょう。リダイレクトは、MS-DOSバージョン2.0に大きく影響を与えたOS、UNIXの標準入出力の考えを取り入れて実現された機能です。2章のファイルシステムのところで、ファイルハンドルについて解説しましたが、ファイルハンドルの0番から4番までの5つは、システムによって使用される予約されたファイルハンドルであったことを覚えているでしょうか。この5つは、**表3.1**のようなデバイスファイル（実際は入出力装置であるが、論理的にはファイルとして扱う）に割り当てられています。

この5つのファイルハンドルは、システムによってあらかじめオープンされており、このハンドルを使っていきなり読み出し、書き込みを行うことができます。またこれらは、入出力が終了したあともクローズする必要はありません。

これらのファイルハンドルは、通常は表3.1に示すデバイス（周辺装置）に割り当てられており、これらのハンドル番号に対して読み出しや書き込みを行えば、それに対応するデバイスに対して読み出しや書き込みが行われることになります。また、表3.1に示されているデバイスに対して入出力を行うシ

ファイルハンドルの番号	名　称	デバイス名	通常の機能（リダイレクトしていない状態）
0	標準入力	CON	コンソールからの入力
1	標準出力	CON	コンソールへの出力
2	標準エラー出力	CON	コンソールへの出力
3	標準補助装置	AUX*	RS-232C の入出力など
4	標準リスト出力	PRN*	プリンタへの出力

COMMAND.COM からはリダイレクトできない

＊バージョン 3.1 以降の PC-9800 シリーズ用 MS-DOS では AUX および PRN デバイスドライバはオプションとなり、CONFIG.SYS に、

```
DEVICE=RSDRV.SYS
DEVICE=PRINT.SYS
```

と指定しておかなければ利用できないものもある。

表 3.1　標準入出力とそのファイルハンドル

ステムコールは、実はそれぞれのファイルハンドルの番号に対して入出力を行っているのです。

2章で述べたように、ファイルハンドルは、システム内に用意されている FCB 領域のうち、どの FCB を使用するかを示す番号でした。しかし、直接 FCB の番号を指すのではなく、実際はファイルハンドル用のテーブルの番号を指しており、そこに FCB 番号が書かれているという間接的なものでした。どうしてこのようにまわりくどいことをするのか、今まで説明はしませんでしたが、実はリダイレクトのためだったのです。

COMMAND.COM は、コマンドラインに「< ファイル名」が書かれていると、そのファイルをオープンし、そのファイルのファイルハンドルを標準入力、つまりファイルハンドルの 0 番に割り当てます。ここで「割り当てる」というのは、ファイルハンドル用のテーブルの 0 番目に、オープンしたファイルの FCB の番号を書き込むことです。たとえば、ファイルをオープンして、ファイルハンドルとして 5 番が返されたとしましょう。この場合、ファイルハンドル用テーブルの 0 番のところへ、ファイルハンドル用テーブルの 5 番の内容をコピーすることにより、標準入力をそのファイルにつなぎ換えるのです。MS-DOS には、このようなファイルハンドルの内容をコピーするシステムコールが用意されており、入出力ストリーム（連続したデータの流れ）の切り換えを必要とするユーザープログラムで利用することができます。

さて、標準入力や標準出力は、通常コンソールに割り当てられています。というより、コンソールに対する入出力が、実は標準入出力に接続されているのです。コンソールから入力するためのシステムコールは、標準入力、つまりファイルハンドルの 0 番から入力を行うので、通常はコンソールから入力が行われることになっています。ここで、ファイルハンドルの 0 番がファイルに接続されているとすると、コンソールからの入力がファイルからの入力に切り換えられることになります。

同様に考えると、ファイルハンドルの 1 番についてもほかのファイルハンドルに接続することによって、リダイレクトを実現することができます。コマンドラインに「> ファイル名」が書かれてい

れば、COMMAND.COMはファイルを書き込みモードでオープンし、そのファイルハンドルを1番にコピーします。この状態を**図3.20**に示します。もし「>> ファイル名」であるなら、ファイルをアペンドモードでオープンし、そのファイルハンドルを1番にコピーします。こうすることにより、コンソールへの出力がそのファイルへリダイレクトされるのです。

なお、標準入力や標準出力がリダイレクトされているときでも、必要があれば、通常のファイルアクセスと同じようにCONをオープンすることによってコンソールと入出力を行うことが可能です。

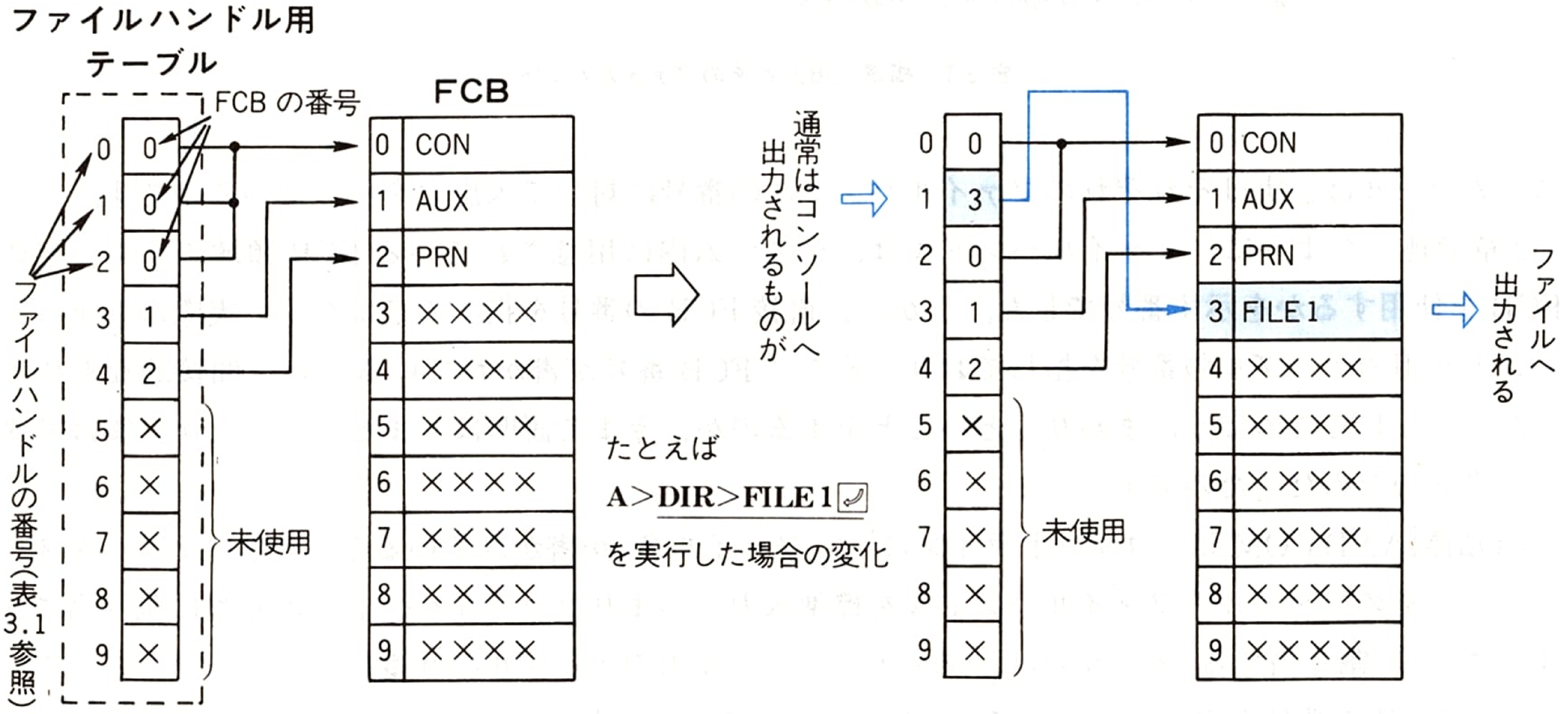

ファイルハンドル用
テーブル
FCBの番号
FCB
ファイルハンドルの番号(表3.1参照)
0
1
2
3
4
5
6
7
8
9
0
0
0
1
2
×
未使用
CON
AUX
PRN
××××
たとえば
A>DIR>FILE1
を実行した場合の変化
通常はコンソールへ出力されるものが
3
FILE1
ファイルへ出力される

図3.20　リダイレクトによるファイルハンドルのつなぎ換え

さて、リダイレクトの仕組みが、かなり明らかになってきたと思います。ファイルハンドルの0番から4番までに割り当てられている5つのデバイスへの入出力は、実際にはそのファイルハンドルを通して行われているために、入力先や出力先を変更することが可能になるのです。それも、プログラムをまったく変更することなく、プログラムを実行する際に、そのコマンドラインでファイルハンドルをつなぎ換える指定をすればよいのですから、たいへん便利です。

このように、入力先や出力先を固定せず、実行時に自由に切り換えてプログラムやコマンドの汎用性を高める考え方を、**標準入出力**と呼んでいます。

■ パイプ

標準入出力は、リダイレクト以外にもパイプという重要な機能を提供します。

パイプは、次のようなコマンドラインによって、「コマンド1」の出力を「コマンド2」に入力することを実現します。

```
A>コマンド1 ［パラメータ］ ｜ コマンド2 ［パラメータ］⏎
```

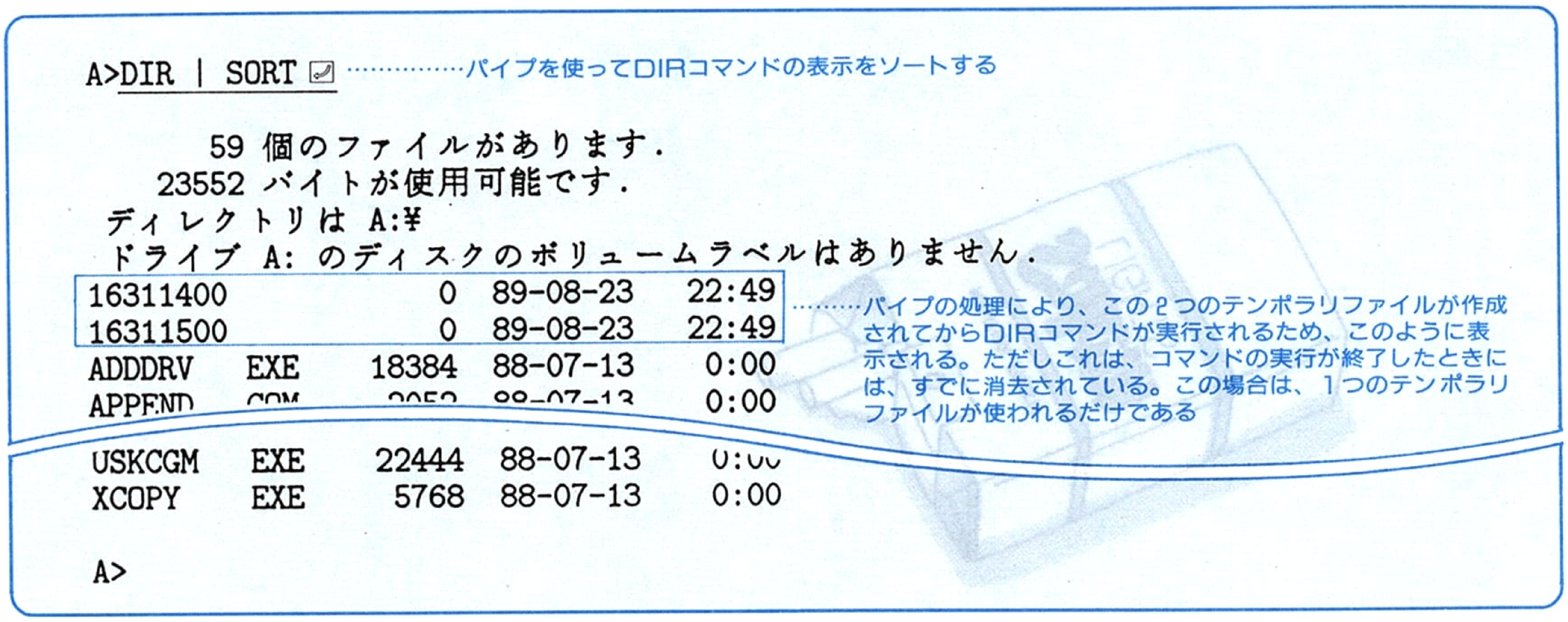
（たとえば、「A>DIR｜SORT⏎など）

UNIXでは、この2つのコマンドが同時に実行され、最初のコマンドの出力が同時に2番目のコマンドに入力されるため、効率よく高速に処理されます。しかし、MS-DOSはシングルタスク（同時には1つの仕事しかできない）のOSであるため、複数のコマンドを同時に実行することができません。このため、MS-DOSのパイプ機能は、リダイレクトの変形として実現されています。

COMMAND.COMは、パイプ処理を実現するために、さきに示したコマンドラインを次のように2つのコマンドに分け、これを連続して実行します。

①**コマンド1 ［パラメータ］ > テンポラリファイル** ……… たとえばA>DIR > TEMP⏎
②**コマンド2 ［パラメータ］ < テンポラリファイル** ……… たとえばA>SORT < TEMP⏎

コマンド1の出力は、リダイレクトによってテンポラリファイル（COMMAND.COM内部で使用するために一時的に作られるファイル）にバッファリング（一時収容）され、コマンド2の入力は、やはりリダイレクトによって同じテンポラリファイルから行われます（**図3.21**）。このように、パイプの処理は、ディスク上のテンポラリファイルを介して行われるので、実行速度はフロッピーディスクではあまり満足できるものではありませんが、ハードディスクやRAMディスクであれば十分速く、たいへん便利に使うことができます。



```
A>DIR | SORT⏎

      59 個のファイルがあります.
   23552 バイトが使用可能です.
 ディレクトリは A:¥
 ドライブ A: のディスクのボリュームラベルはありません.
16311400                0  89-08-23   22:49
16311500                0  89-08-23   22:49
ADDDRV     EXE      18384  88-07-13    0:00
APPEND     COM       2052  88-07-13    0:00

USKCGM     EXE      22444  88-07-13    0:00
XCOPY      EXE       5768  88-07-13    0:00

A>
```

……パイプを使ってDIRコマンドの表示をソートする

……パイプの処理により、この2つのテンポラリファイルが作成されてからDIRコマンドが実行されるため、このように表示される。ただしこれは、コマンドの実行が終了したときには、すでに消去されている。この場合は、1つのテンポラリファイルが使われるだけである

図3.21　「A>DIR ｜ SORT」の実行結果に現れるテンポラリファイル

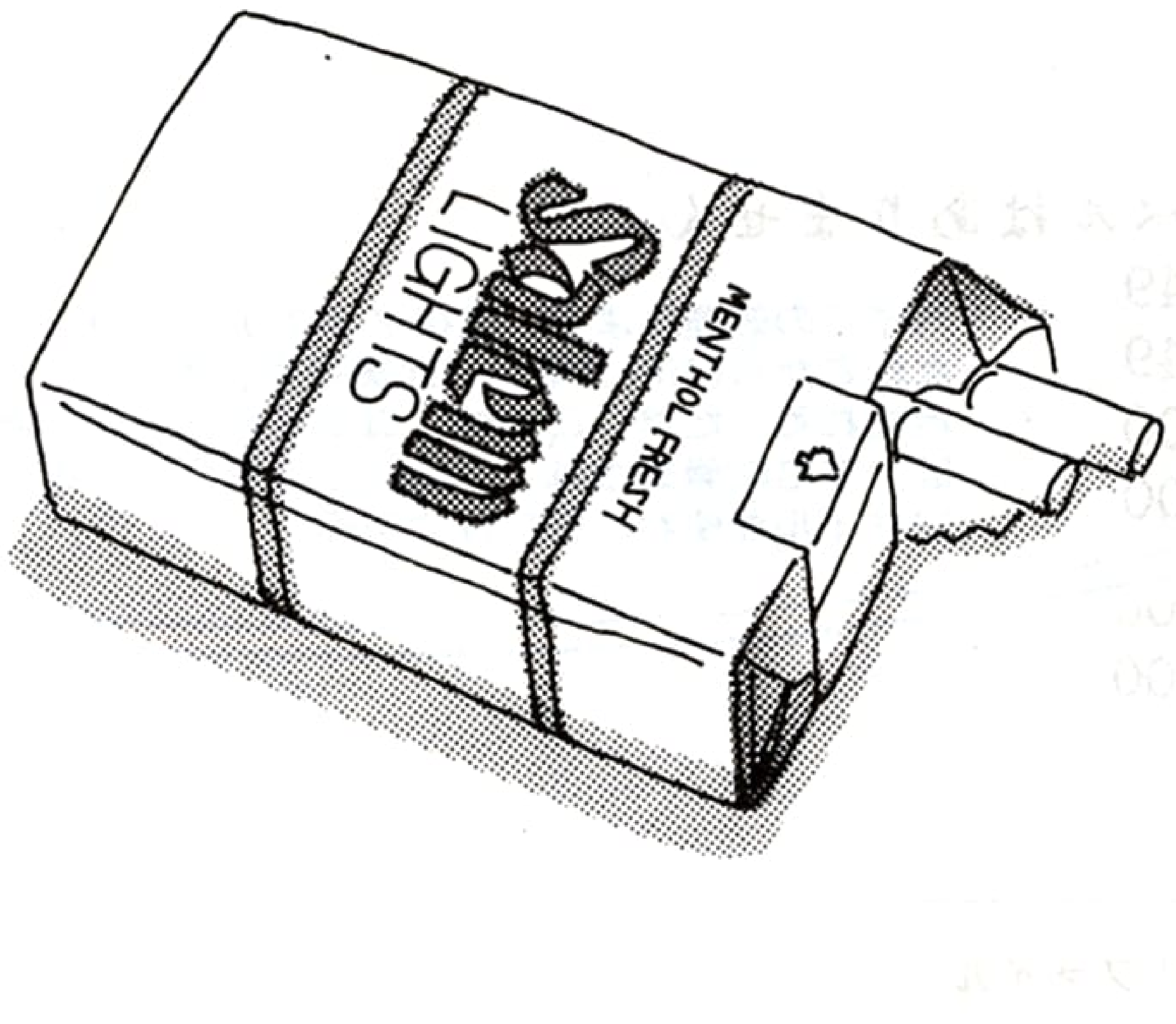
■ フィルタ

標準入出力の考え方は、パイプやリダイレクトを実現したり、コマンド・プロセッサなどの構成やそのプログラムが簡単になるだけでなく、さらに高い汎用性を持つというすばらしいアイデアです。リダイレクトやパイプによって、入力先や出力先に指定されたファイルのオープン／クローズは、COMMAND.COMによって行われるので、実行する外部プログラムや内蔵コマンドでは何も考慮する必要はありません。ユーザープログラムを作る際に「標準入力から読み込み、標準出力に書き出す」ようにしておけば、実行時にリダイレクトを使ってファイルへの入出力ができるわけで、そのファイルのオープンやクローズといったことはまったく考える必要はありません。また、リダイレクトを使って、出力先をPRNにすることにより、プリンタに出力することも簡単にできてしまいます。このような標準入出力を使って作られたプログラムのことを**フィルタ**と呼びます。

UNIXでは、1つひとつのコマンドは非常に単純な機能なのに、そのコマンドをパイプで何段にも接続して実行することにより、非常に複雑な処理を行わせることができます。MS-DOSには標準のフィルタとして、みなさんもよくご存じの次に示す3つのコマンドが用意されています。

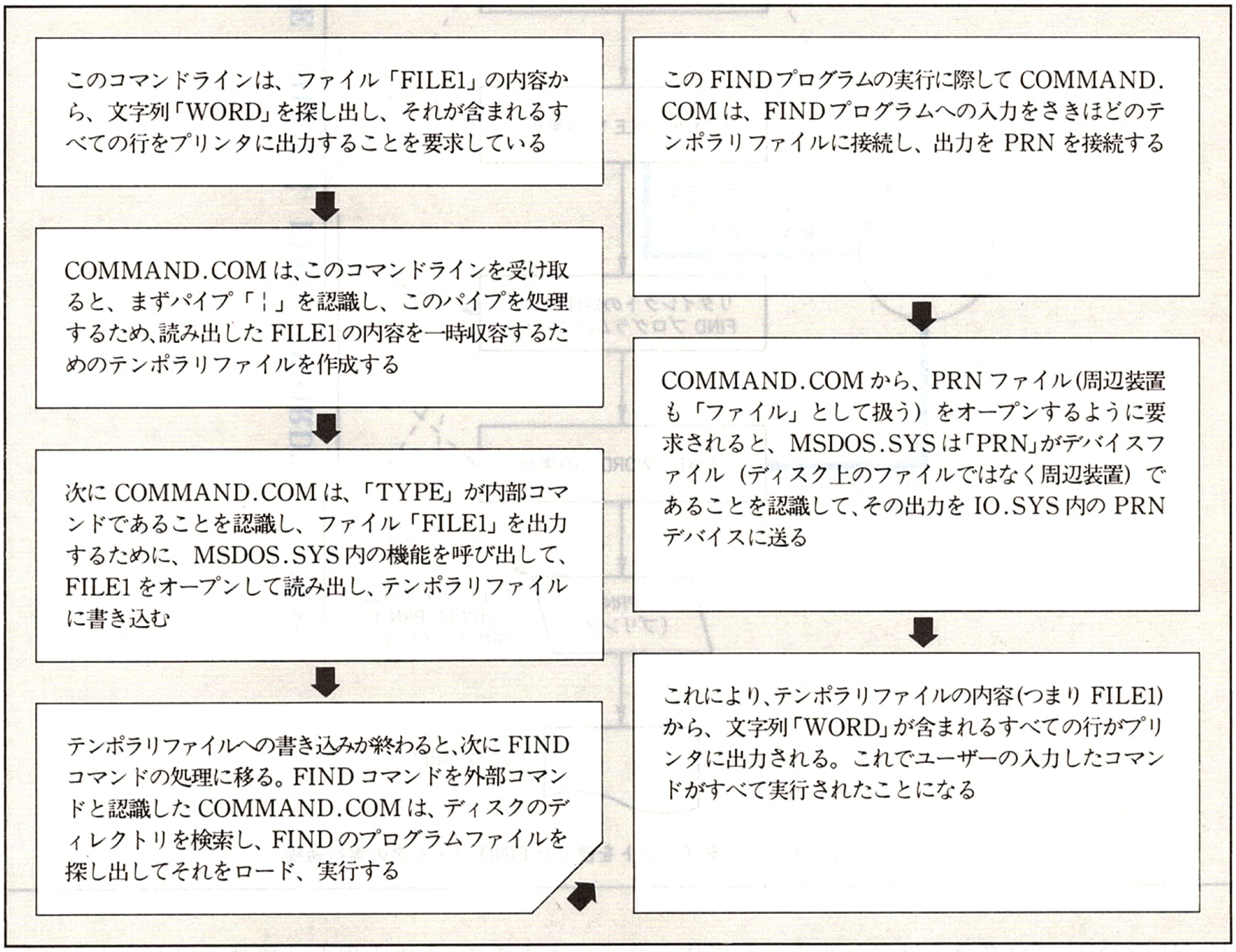
MORE………ディスプレイ出力を1画面ごとに表示する
FIND………文字列を検索する
SORT………文字列をソートする

これらのコマンドによっては、COMMAND.COMがファイルをオープンしたり、テンポラリファイルを作成したりします。COMMAND.COMによるコマンドラインの解釈には、リダイレクトやパイプも含まれますので、それらの記号「<」「>」「>>」「|」がコマンドラインに含まれていれば、COMMAND.COMはそれに応じた処理を行います。



では、例として、ユーザーの入力した次のFINDフィルタのコマンドがどのように処理されるかを見ていきましょう(図3.22参照)。

```
A>TYPE  FILE1  |  FIND  ''WORD''  >  PRN↵
```



このコマンドラインは、ファイル「FILE1」の内容から、文字列「WORD」を探し出し、それが含まれるすべての行をプリンタに出力することを要求している

COMMAND.COMは、このコマンドラインを受け取ると、まずパイプ「¦」を認識し、このパイプを処理するため、読み出したFILE1の内容を一時収容するためのテンポラリファイルを作成する

次にCOMMAND.COMは、「TYPE」が内部コマンドであることを認識し、ファイル「FILE1」を出力するために、MSDOS.SYS内の機能を呼び出して、FILE1をオープンして読み出し、テンポラリファイルに書き込む

テンポラリファイルへの書き込みが終わると、次にFINDコマンドの処理に移る。FINDコマンドを外部コマンドと認識したCOMMAND.COMは、ディスクのディレクトリを検索し、FINDのプログラムファイルを探し出してそれをロード、実行する

このFINDプログラムの実行に際してCOMMAND.COMは、FINDプログラムへの入力をさきほどのテンポラリファイルに接続し、出力をPRNを接続する

COMMAND.COMから、PRNファイル(周辺装置も「ファイル」として扱う)をオープンするように要求されると、MSDOS.SYSは「PRN」がデバイスファイル(ディスク上のファイルではなく周辺装置)であることを認識して、その出力をIO.SYS内のPRNデバイスに送る

これにより、テンポラリファイルの内容(つまりFILE1)から、文字列「WORD」が含まれるすべての行がプリンタに出力される。これでユーザーの入力したコマンドがすべて実行されたことになる

— 次ページ図3.22に続く —

また、このフィルタについては、自分で作成してみるのも、プログラム開発のよい例題になります。たとえば、テキストファイル(漢字が含まれていてもよい)の大文字を小文字に、あるいは小文字を大文字に変換するフィルタなどはどうでしょうか。もちろん漢字はそのままで変化しないようなプログラムにしなければなりません。これなどは実用性もあり、おもしろいフィルタのひとつですので参考までに作成してみましょう。

このプログラム名を「ULCONV」(Upper Lower CONVerter)として、その機能を示します。な

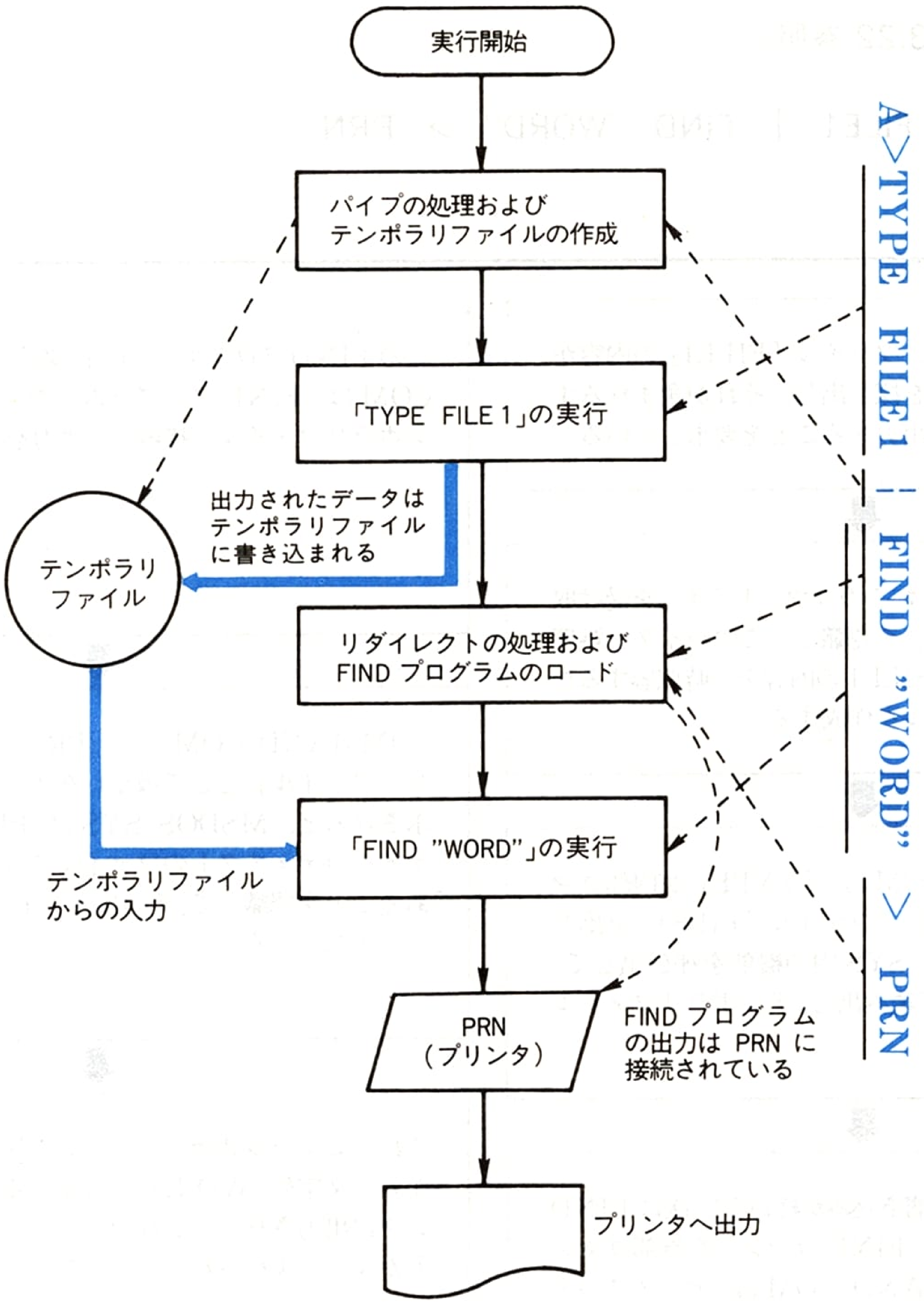

実行開始
パイプの処理および
テンポラリファイルの作成
「TYPE　FILE 1」の実行
出力されたデータは
テンポラリファイル
に書き込まれる
テンポラリ
ファイル
リダイレクトの処理および
FIND プログラムのロード
「FIND　"WORD"」の実行
テンポラリファイル
からの入力
PRN
（プリンタ）
FIND プログラム
の出力は PRN に
接続されている
プリンタへ出力
A>TYPE　FILE1　¦　FIND　"WORD"　>　PRN

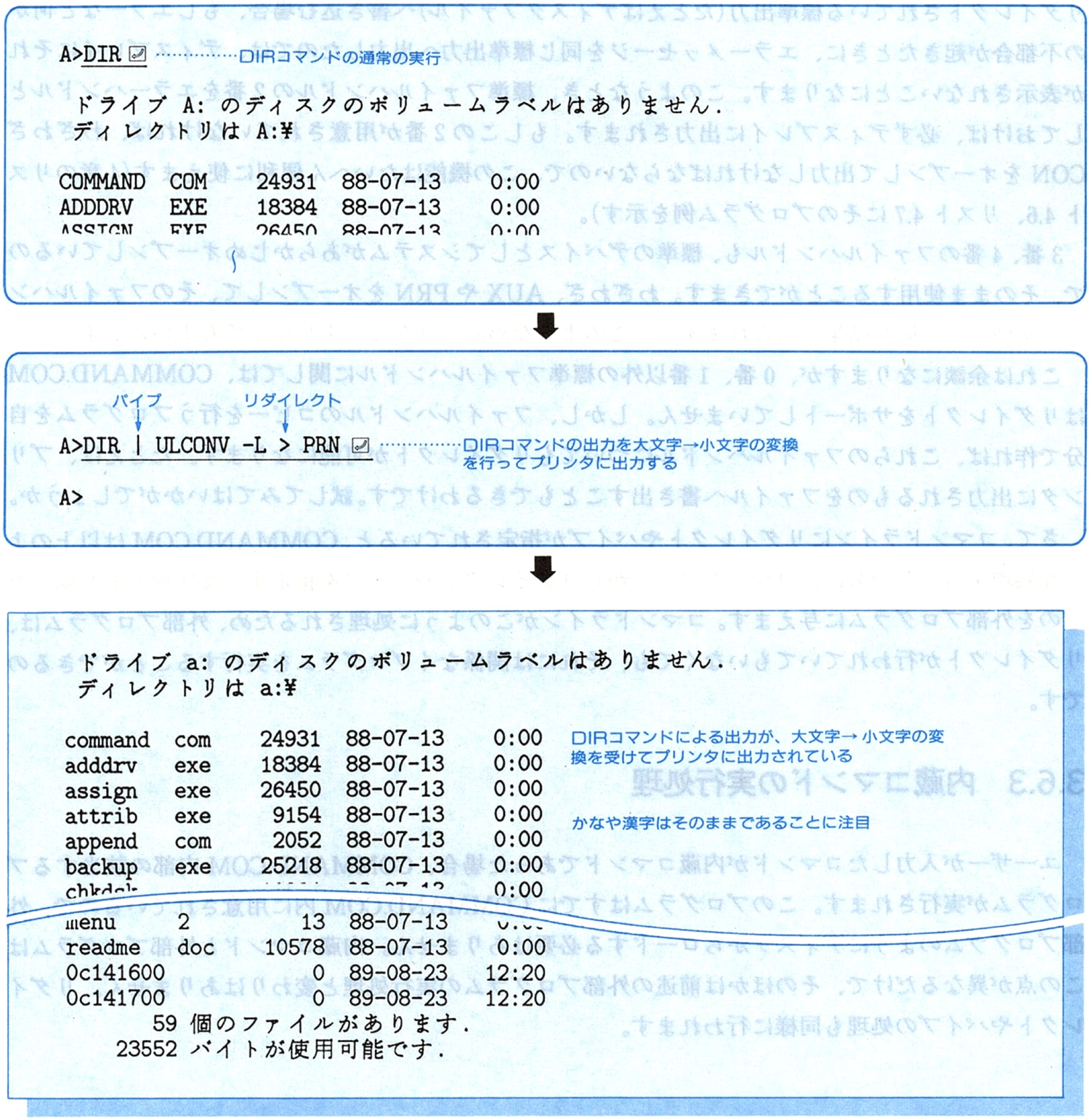
図 3.22　パイプおよびリダイレクトを使った FIND フィルタの実行過程

お、ULCONV プログラムはフィルタとして作られていますので、その入出力は標準入出力によって行います*。

A＞ULCONV　－U ………入力された文字列中の小文字を大文字に変換する。
　　　　　　　　　　ただし漢字はそのまま
A＞ULCONV　－L ………入力された文字列中の大文字を小文字に変換する。
　　　　　　　　　　ただし漢字はそのまま

＊ また、「A>ULCONV －U　ファイル名↵」のように、入力ファイルを指定することもできる。

このフィルタのプログラムは、C 言語を使えば簡単に書けますので、そのソースプログラムを巻末の APPENDIX に示しておきます。ここでは実行可能なプログラム「ULCONV.EXE」が完成したとして、そのフィルタの実行例を示しましょう（**図 3.23**）。実行例のパイプやリダイレクトの使い方にも注目してください。



```
A>DIR ↵

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13     0:00
ADDDRV   EXE     18384  88-07-13     0:00
ASSIGN   EXE     26450  88-07-13     0:00
```

DIRコマンドの通常の実行

パイプ　リダイレクト

```
A>DIR | ULCONV -L > PRN ↵

A>
```

DIRコマンドの出力を大文字→小文字の変換を行ってプリンタに出力する

```
 ドライブ a: のディスクのボリュームラベルはありません.
 ディレクトリは a:¥

command  com     24931  88-07-13     0:00
adddrv   exe     18384  88-07-13     0:00
assign   exe     26450  88-07-13     0:00
attrib   exe      9154  88-07-13     0:00
append   com      2052  88-07-13     0:00
backup   exe     25218  88-07-13     0:00
chkdsk                               0:00
menu     2          13  88-07-13
readme   doc     10578  88-07-13     0:00
0c141600             0  89-08-23    12:20
0c141700             0  89-08-23    12:20
        59 個のファイルがあります.
     23552 バイトが使用可能です.
```

DIRコマンドによる出力が、大文字→小文字の変換を受けてプリンタに出力されている

かなや漢字はそのままであることに注目

図 3.23　ユーザーが作成したフィルタの実行例

さてここで、0番、1番以外の標準ファイルハンドルについても、少し説明を加えておきましょう。

2番の標準エラー出力は、通常CONに割り当てられています。1番の標準出力のほかに、わざわざ同じCONに接続されているファイルハンドルを用意するのは、どういう意味なのでしょう。これは、標準出力がリダイレクトされていて、ディスプレイには出力されないときにも、ディスプレイへメッセージを出力したい場合があるからです。たとえば、フィルタのプログラムなどで、その実行結果を、リダイレクトされている標準出力(たとえばディスクファイル)へ書き込む場合、もしエラーなど何かの不都合が起きたときに、エラーメッセージを同じ標準出力へ出力したのでは、ディスプレイにそれが表示されないことになります。このようなとき、標準ファイルハンドルの2番をエラーハンドルとしておけば、必ずディスプレイに出力されます。もしこの2番が用意されていなければ、わざわざCONをオープンして出力しなければならないので、この機能はたいへん便利に使えます(4章のリスト4.6、リスト4.7にそのプログラム例を示す)。

3番、4番のファイルハンドルも、標準のデバイスとしてシステムがあらかじめオープンしているので、そのまま使用することができます。わざわざ、AUXやPRNをオープンして、そのファイルハンドルを使っても同じ結果が得られますが、このようなめんどうなことをしなくてもよいのです。

これは余談になりますが、0番、1番以外の標準ファイルハンドルに関しては、COMMAND.COMはリダイレクトをサポートしていません。しかし、ファイルハンドルのコピーを行うプログラムを自分で作れば、これらのファイルハンドルについてもリダイレクトが可能になります。たとえば、プリンタに出力されるものをファイルへ書き出すこともできるわけです。試してみてはいかがでしょうか。

さて、コマンドラインにリダイレクトやパイプが指定されていると、COMMAND.COMは以上のような処理を行ったのちに、コマンドラインからリダイレクトやパイプを指定する文字列を取り除いたものを外部プログラムに与えます。コマンドラインがこのように処理されるため、外部プログラムは、リダイレクトが行われていてもいなくても、それには関係なくプログラムを実行することができるのです。

3.6.3　内蔵コマンドの実行処理

ユーザーが入力したコマンドが内蔵コマンドであった場合、COMMAND.COM内部の該当するプログラムが実行されます。このプログラムはすでにCOMMAND.COM内に用意されているので、外部プログラムのようにディスクからロードする必要はありません。内蔵コマンドと外部プログラムはこの点が異なるだけで、そのほかは前述の外部プログラムの実行処理と変わりはありません。リダイレクトやパイプの処理も同様に行われます。

■ 環境変数の設定

内蔵コマンドのうち、環境変数の設定については、とくに触れておく必要があるでしょう。環境変数は、その機能と使い方を理解すれば、MS-DOSになくてはならないものであることがわかります。

環境変数に文字列を設定する、つまり環境を設定するにはSETコマンドを使います。COMMAND.COM以外の標準のコマンドで、環境変数を有効に活用したものはなく、市販の各種のソフトウェアも、初期の頃はあまり環境を意識したものはありませんでした。ところがMS-DOSバージョン3.*x*のコマンドや、最近の市販ソフトウェアでは、この環境変数を積極的に利用するようになり、その効果を上げています。

環境変数は、SETコマンドで次のように設定します(図3.24参照)。

SET　環境変数名＝文字列　(設定する環境変数名や文字列は任意)

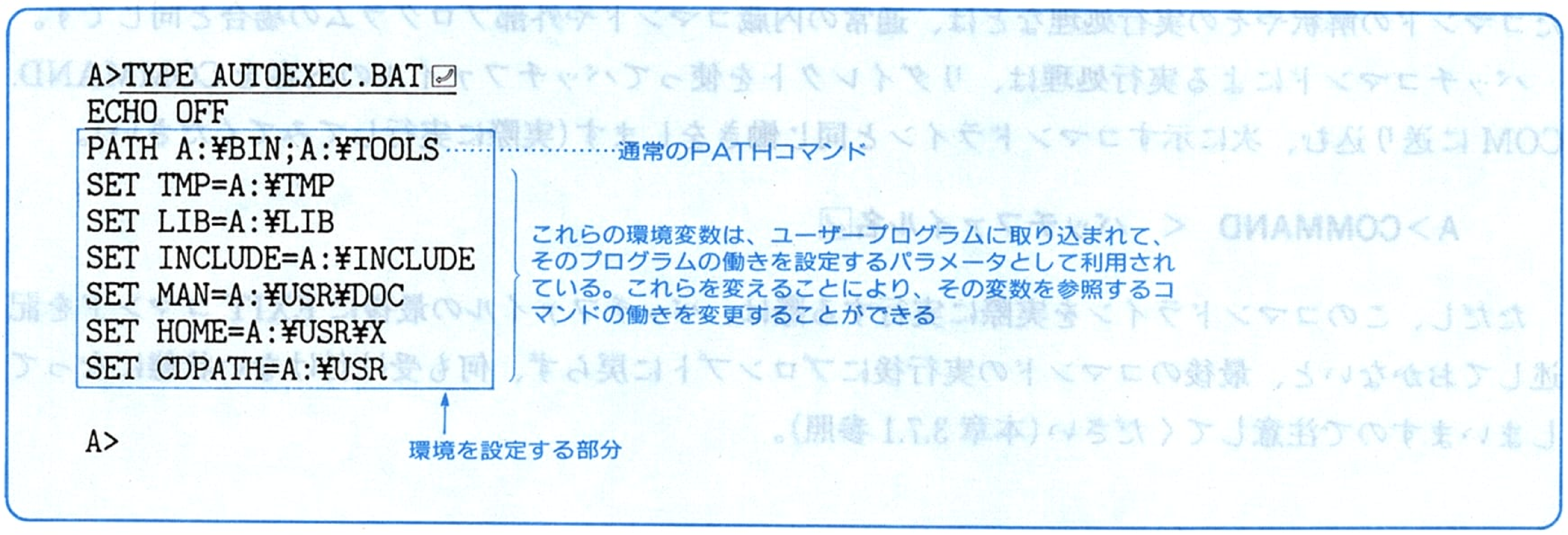
ただし環境変数PATHとPROMPTを設定するには、SETを省くこともできます(というより普通は省き、「PATH　文字列」のように設定する)。

SETコマンドで、環境変数に文字列を設定した行「環境変数名＝文字列」の集まりを、MS-DOSの**環境**と呼び、この行「環境変数名＝文字列」自身を**環境文字列**と呼びます。MS-DOSの環境は、この環境文字列が集まったものであり、重要なのは、この環境を取り込んで自由に利用することができるということです。このことは、環境の設定さえ変えれば、ユーザープログラムを変更することなく、ユーザープログラムの機能を変えることが可能なことを意味しています。

また、設定した環境変数を削除するには、SETコマンドを次のように実行します。

SET　環境変数名＝

図3.24に、環境を設定する実例を示しましょう。これは、一般的なプログラム開発をするのに必要な環境を設定したもので、AUTOEXEC.BATファイルに登録してあります。



```
A>TYPE AUTOEXEC.BAT↵
ECHO OFF
PATH A:¥BIN;A:¥TOOLS
SET TMP=A:¥TMP
SET LIB=A:¥LIB
SET INCLUDE=A:¥INCLUDE
SET MAN=A:¥USR¥DOC
SET HOME=A:¥USR¥X
SET CDPATH=A:¥USR

A>
```

通常のPATHコマンド

これらの環境変数は、ユーザープログラムに取り込まれて、そのプログラムの働きを設定するパラメータとして利用されている。これらを変えることにより、その変数を参照するコマンドの働きを変更することができる

環境を設定する部分

図3.24　環境変数の設定例

環境変数は、次のようにメモリ上に環境文字列が並んだ形で格納されます。

環境変数＝文字列

この環境文字列がどのように外部プログラムに渡されるのかは後述しますが、環境文字列は、リダイレクトやパイプと同じように、プログラムの汎用性を高めるものとして重要です。つまり外部プログラムは、環境変数を参照することによって、プログラムを書き換えることなくプログラムの仕様を変更することができるのです(外部プログラムが、そのようにプログラミングしてあれば)。

COMMAND.COMは、外部コマンドのサーチパスやプロンプト文字列を、環境変数PATHおよびPROMPTを参照して決めているため、その変更を動的に行うことができます。また、標準のリンカLINKのバージョン3.0以降(Microsoft Cや、MS-DOSバージョン3.*x*に付属)は、環境変数LIBによってライブラリのサーチパスを決定するため、従来のようにいちいちライブラリ名をフルパス名で指定する必要はありません。あらかじめ、

```
SET  LIB＝¥LIB
```

などのように設定しておくだけでよいのです。

市販のソフトウェアの中にも、環境文字列を参照して各種パラメータを決定しているものが多くなってきましたが、さらに多くのソフトウェアで有効に活用してほしいものです。また、あなたが作成するプログラムでも、この環境変数をうまく利用すれば、より使いやすいものになるでしょう。

3.6.4　バッチコマンドの処理

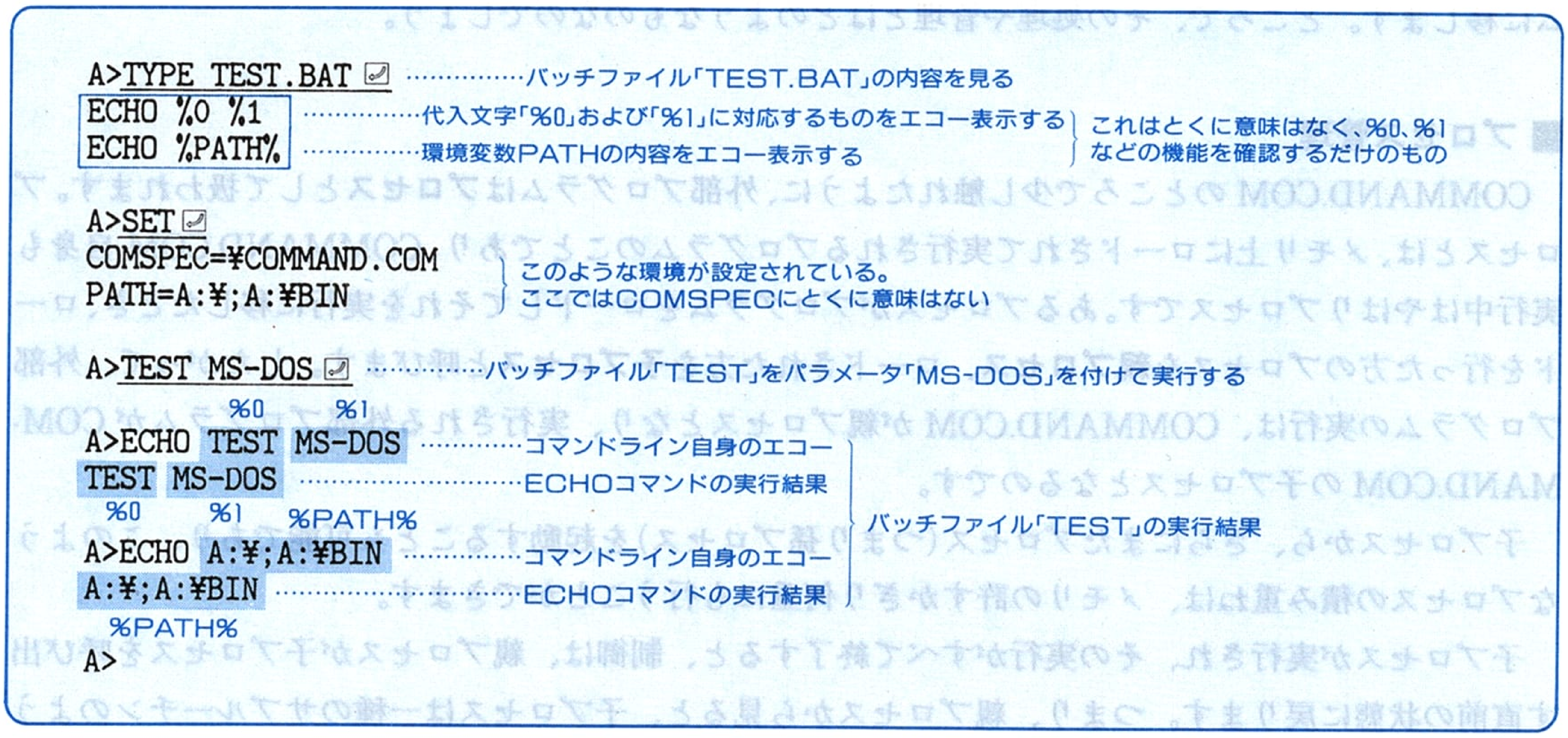
バッチコマンドは、いくつかのコマンドを並べたバッチファイルを作り、そのファイル名によって、ファイル内容の一連のコマンドを順次自動的に実行するものです。具体的には、通常はコンソールから入力されるコマンドが、ディスク上のバッチファイルから入力されることになるだけで、入力されたコマンドの解釈やその実行処理などは、通常の内蔵コマンドや外部プログラムの場合と同じです。

バッチコマンドによる実行処理は、リダイレクトを使ってバッチファイルの内容をCOMMAND.COMに送り込む、次に示すコマンドラインと同じ働きをします(実際に実行してみてください)。

```
A＞COMMAND  ＜  バッチファイル名⏎
```

ただし、このコマンドラインを実際に実行する際は、バッチファイルの最後にEXITコマンドを記述しておかないと、最後のコマンドの実行後にプロンプトに戻らず、何も受け付けない状態になってしまいますので注意してください(本章3.7.1参照)。

バッチコマンドの実行において、通常のコマンドラインの解釈と違うのは、バッチコマンドではパラメータの置き換えが行われることです。バッチファイル中に「%0」〜「%9」という文字列があると、そのバッチファイルを実行するコマンドラインの文字列と置き換えられます。「%0」はコマンド名、「%1」が最初のパラメータ、「%2」が2番目のパラメータという具合いです。また、「%環境変数%」と書くと、環境変数の右辺の文字列に置き換えられます(図3.25参照)。



```
A>TYPE TEST.BAT ⏎
ECHO %0 %1
ECHO %PATH%

A>SET ⏎
COMSPEC=¥COMMAND.COM
PATH=A:¥;A:¥BIN

A>TEST MS-DOS ⏎

A>ECHO TEST MS-DOS
TEST MS-DOS

A>ECHO A:¥;A:¥BIN
A:¥;A:¥BIN

A>
```

バッチファイル「TEST.BAT」の内容を見る

代入文字「%0」および「%1」に対応するものをエコー表示する

環境変数PATHの内容をエコー表示する

これはとくに意味はなく、%0、%1などの機能を確認するだけのもの

このような環境が設定されている。ここではCOMSPECにとくに意味はない

バッチファイル「TEST」をパラメータ「MS-DOS」を付けて実行する

コマンドライン自身のエコー

ECHOコマンドの実行結果

コマンドライン自身のエコー

ECHOコマンドの実行結果

バッチファイル「TEST」の実行結果

図3.25　バッチファイルの代入文字列の機能の確認

3.7　MSDOS.SYS の働き

MSDOS.SYS は MS-DOS そのものであり、MS-DOS の能力は MSDOS.SYS の能力によるといってもよいでしょう。MS-DOS と私たちとをインターフェイスし、MS-DOS の顔の役目を果たしているのが、コマンド・プロセッサの COMMAND.COM であるため、初心者には、COMMAND.COM の機能や使い勝手が MS-DOS の能力のように受け取られがちですが、これは MS-DOS 上の1つのプログラムにすぎません。また、この機能を支えているのも、MSDOS.SYS であることは言うまでもないでしょう。リダイレクトやパイプ、それに環境といった UNIX の思想を取り入れた便利な機能も、MSDOS.SYS のサポートによって実現されているのです。

本節では、MS-DOS の中心である MSDOS.SYS が、外部プログラムや内蔵コマンドの実行にどのような役割を果たしているのかを見ていきましょう。

3.7.1　外部プログラムの実行とMSDOS.SYS

ユーザーが入力したコマンドラインをCOMMAND.COMが解釈し、それが外部プログラムの実行の要求である場合は、MSDOS.SYSはディスクから目的のプログラムファイルを読み込み、そのプログラムを実行するために必要となる、各種の処理や情報の管理を行ったあと制御権を目的のプログラムに移します。ところで、その処理や管理とはどのようなものなのでしょう。

■プロセス管理

COMMAND.COMのところで少し触れたように、外部プログラムは**プロセス**として扱われます。プロセスとは、メモリ上にロードされて実行されるプログラムのことであり、COMMAND.COM自身も実行中はやはりプロセスです。あるプロセスがプログラムをロードしてそれを実行に移したとき、ロードを行った方のプロセスを**親プロセス**、ロードされた方を**子プロセス**と呼びます。したがって、外部プログラムの実行は、COMMAND.COMが親プロセスとなり、実行される外部プログラムがCOMMAND.COMの子プロセスとなるのです。

子プロセスから、さらにまたプロセス(つまり孫プロセス)を起動することも可能であり、このようなプロセスの積み重ねは、メモリの許すかぎり何重にも行うことができます。

子プロセスが実行され、その実行がすべて終了すると、制御は、親プロセスが子プロセスを呼び出す直前の状態に戻ります。つまり、親プロセスから見ると、子プロセスは一種のサブルーチンのようなものと考えることができます。

子プロセスを起動するには、プログラム名(パス名)のほかに、そのコマンドラインの文字列や、環境文字列などが必要です。プロセスを起動するのに必要なこれらの情報は、コマンドラインの文字列と環境文字列によって親プロセスから子プロセスへ引き渡されます。たとえば、親プロセスがオープンしたファイルは、そのまま子プロセスに引き継がれますので、子プロセスにおいても、親プロセスのときと同じファイルハンドルを使って、オープンした状態のままのそのファイルをアクセスすることができます。この仕組みは具体的には、リダイレクトによって標準入出力が受け継がれることによって実現されています。

MSDOS.SYSのプロセス管理によって、このように親プロセスから子プロセスへは各種の情報が引き渡されますが、子プロセスから親プロセスへ戻るときには別の処理が行われます。親プロセスの環境やオープンしたファイルは、子プロセスに引き継がれるのですが、子プロセスが環境を変更したり、ファイルをクローズしたりしても、親プロセスに戻るときにそれらは引き継がれません。つまり、親プロセスの環境は変化せず、ファイルもクローズされないのです(図3.26)。

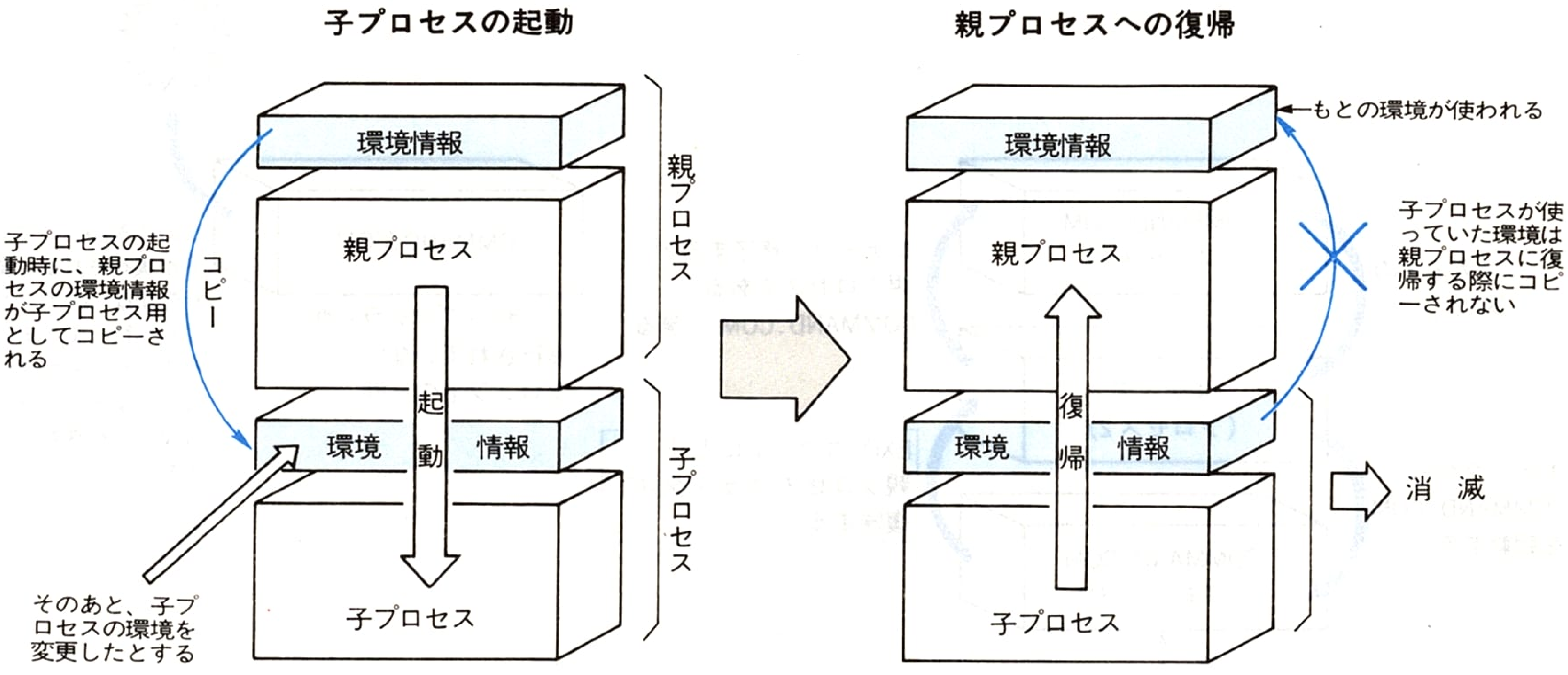

子プロセスの起動
親プロセスへの復帰
環境情報
親プロセス
起動
環境
情報
子プロセス
親プロセス
子プロセス
コピー
子プロセスの起動時に、親プロセスの環境情報が子プロセス用としてコピーされる
そのあと、子プロセスの環境を変更したとする
もとの環境が使われる
子プロセスが使っていた環境は親プロセスに復帰する際にコピーされない
復帰
消　滅

図 3.26　親プロセスと子プロセスの関係、および親から子への環境の引き継ぎ

このような、親プロセスから子プロセスを実行できるプログラムの実例が、4章のファンクション4A_H、4B_H、4C_H(リスト 4.9)にありますので参照してください。

さて、さきの 3.2.4 では、MS-DOS の起動時に COMMAND.COM は COMMAND.COM をシステムに常駐させるために/P オプション付きでロードされると説明しましたが、この「常駐」の意味についてもう少し詳しく解説しましょう。

エディタやビジネスソフトなどで、実行中に子プロセスを起動することができるものがありますが、たとえばそこで子プロセスとして COMMAND.COM を起動し、その内蔵コマンドを実行したとしましょう。必要な内蔵コマンドを実行して目的を達成したあと、COMMAND.COM からもとのエディタには EXIT コマンドで戻ることができます。つまり、EXIT コマンドを実行すると親プロセスに戻るのです。ところが、MS-DOS の起動時にロードされる COMMAND.COM には、戻るべき親プロセスがないので(起動している COMMAND.COM 自身が親プロセスである)、EXIT コマンドが実行されては COMMAND.COM を飛び出してどこかへ行ってしまうことになります。そこで/P オプションを付けることにより、EXIT コマンドによる親プロセスに戻る機能を抑止しておくのです。ですから、COMMAND.COM を再ロードするリブートパスを変更するために、CONFIG.SYS ファイルに SHELL コマンドを登録したときには、必ずこの/P オプションの指定を忘れないようにしなければなりません。(図 3.27 参照)

また/P オプションは、システム起動時に初めて起動されるコマンドプロセッサであることを意味しますので、このオプションがあると、AUTOEXEC.BAT が実行されます。

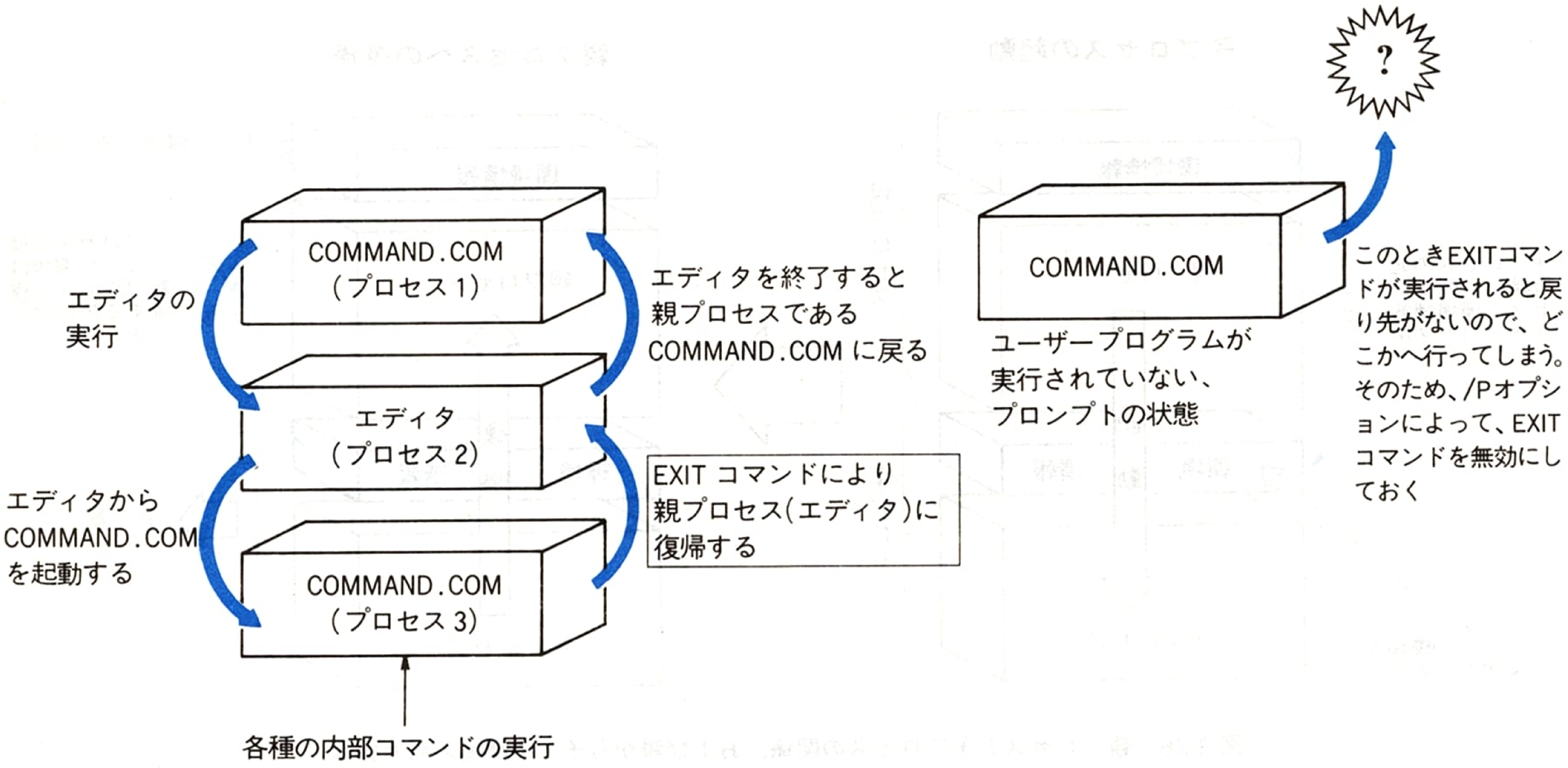

COMMAND.COM
（プロセス 1）
エディタの
実行
エディタを終了すると
親プロセスである
COMMAND.COM に戻る
エディタ
（プロセス 2）
エディタから
COMMAND.COM
を起動する
EXIT コマンドにより
親プロセス（エディタ）に
復帰する
COMMAND.COM
（プロセス 3）
各種の内部コマンドの実行
?
COMMAND.COM
ユーザープログラムが
実行されていない、
プロンプトの状態
このときEXITコマンドが実行されると戻り先がないので、どこかへ行ってしまう。
そのため、/Pオプションによって、EXITコマンドを無効にしておく

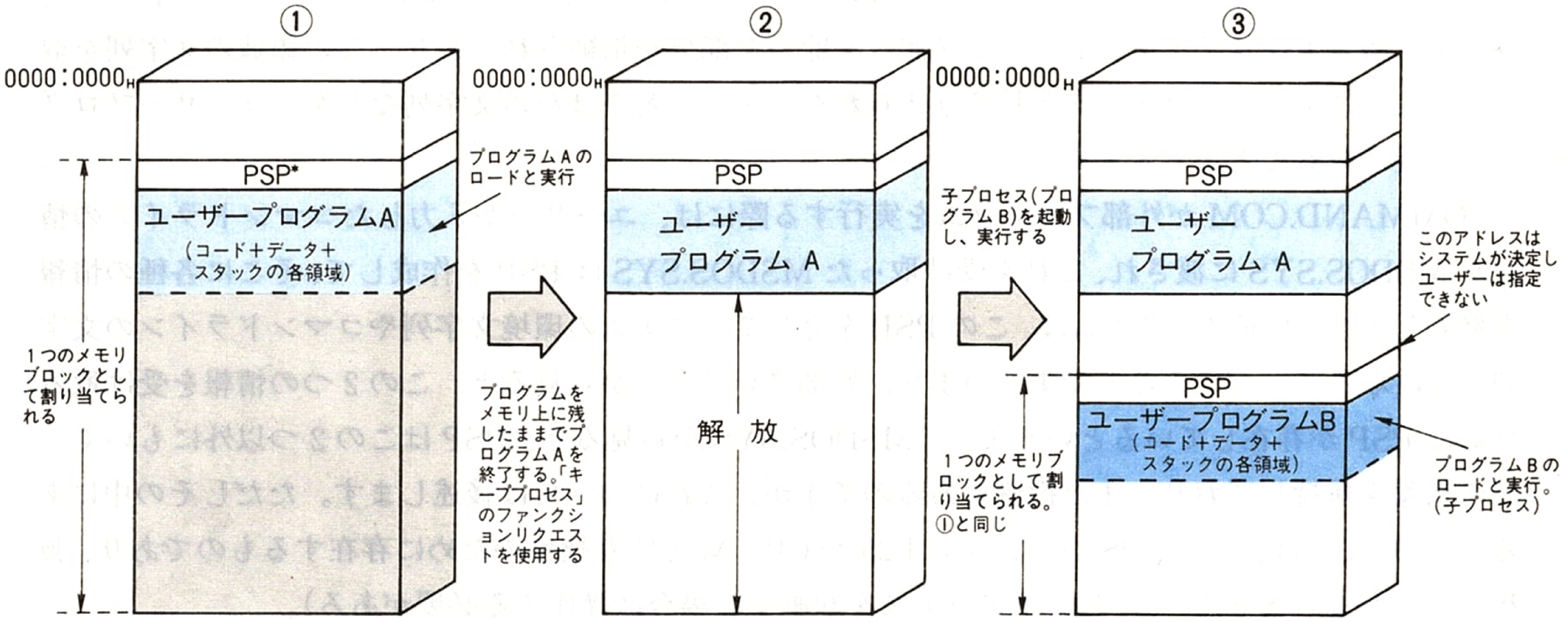
図 3.27　COMMAND.COM の/P オプション

さて以上のように、親プロセスから子プロセスが起動できるという意味は、現在のプロセスから外部プログラムをサブルーチンのように実行することを可能にするものです。この「プロセス」は、一般にはマルチタスクの OS において、同時に実行される複数のプログラムの 1 つひとつを指す言葉であり、次世代の OS である OS/2 では、子プロセスの実行終了を待たずに、親プロセスが並行して実行を続けることができます。

■ メモリ管理

MS-DOS では、限られたメモリ領域を各プロセスに効率よく割り当てるために、メモリの使用もシステムの管理下に置かれます。もちろんこの管理は MSDOS.SYS の役目です。

各プロセスには（それぞれの外部プログラムの実行には）任意のサイズのメモリブロック（最小は 16 バイト単位）を割り当てることができ、その範囲を自由に使用することができます。また、割り当てられたメモリのサイズを変更したり、新たなメモリブロックを取得したり、使用後にそれらを解放したりすることが可能です（これらの機能はシステムコールに用意されている）。ただし、子プロセスを起動するには、メモリにそのための十分な空き領域が必要であり、十分な領域が確保できない場合は、その親プロセスが占有しているメモリブロックの一部を、子プロセスのために解放しなければならない場合もあります。子プロセスが終了すると、そのプロセスが占有していたメモリブロックは解放され、ほかのプロセスで使用することが可能になります（**図 3.28** 参照）。このようなメモリ管理を利用し


①
②
③
0000:0000H
0000:0000H
0000:0000H
PSP*
ユーザープログラムA
(コード+データ+スタックの各領域)
プログラムAのロードと実行
1つのメモリブロックとして割り当てられる
プログラムをメモリ上に残したままでプログラムAを終了する。「キーププロセス」のファンクションリクエストを使用する
PSP
ユーザープログラム A
解　放
子プロセス(プログラムB)を起動し、実行する
PSP
ユーザープログラム A
このアドレスはシステムが決定しユーザーは指定できない
PSP
ユーザープログラムB
(コード+データ+スタックの各領域)
1つのメモリブロックとして割り当てられる。①と同じ
プログラムBのロードと実行。(子プロセス)

最初にロードして実行するプログラムは、PSPの先頭からメモリの最後部までが1つのメモリブロックとして割り当てられている

割り当てられたメモリブロックのサイズを変更するファンクションリクエストを使って、ユーザープログラムAで使用されないメモリ領域を解放する

次にプログラムBをロードして、プログラムAの子プロセスとして実行する。プログラムAに戻ったときは、②と同様に、プログラムBで使われていたメモリ領域を解放しておく

＊PSP については後述

図 3.28　メモリ管理の概念

たプログラムの実例が、さきの子プロセスの起動と同じプログラム(リスト 4.9)にありますので参照してください。

■ PSP(プログラムセグメント・プレフィックス)

外部プログラムを実行する際には、メモリ上に必ず **PSP**(Program Segment Prefix)というものが作られます。この PSP は、MS-DOS のシステムが、外部プログラムに対して種々の情報を引き渡したり、システム自身がそのプロセスの情報を保持するために使われます。

ユーザープログラムから見て意味のある PSP の情報は次の 2 つです。

- ●環境セグメントアドレス
- ●コマンドラインのパラメータ文字列

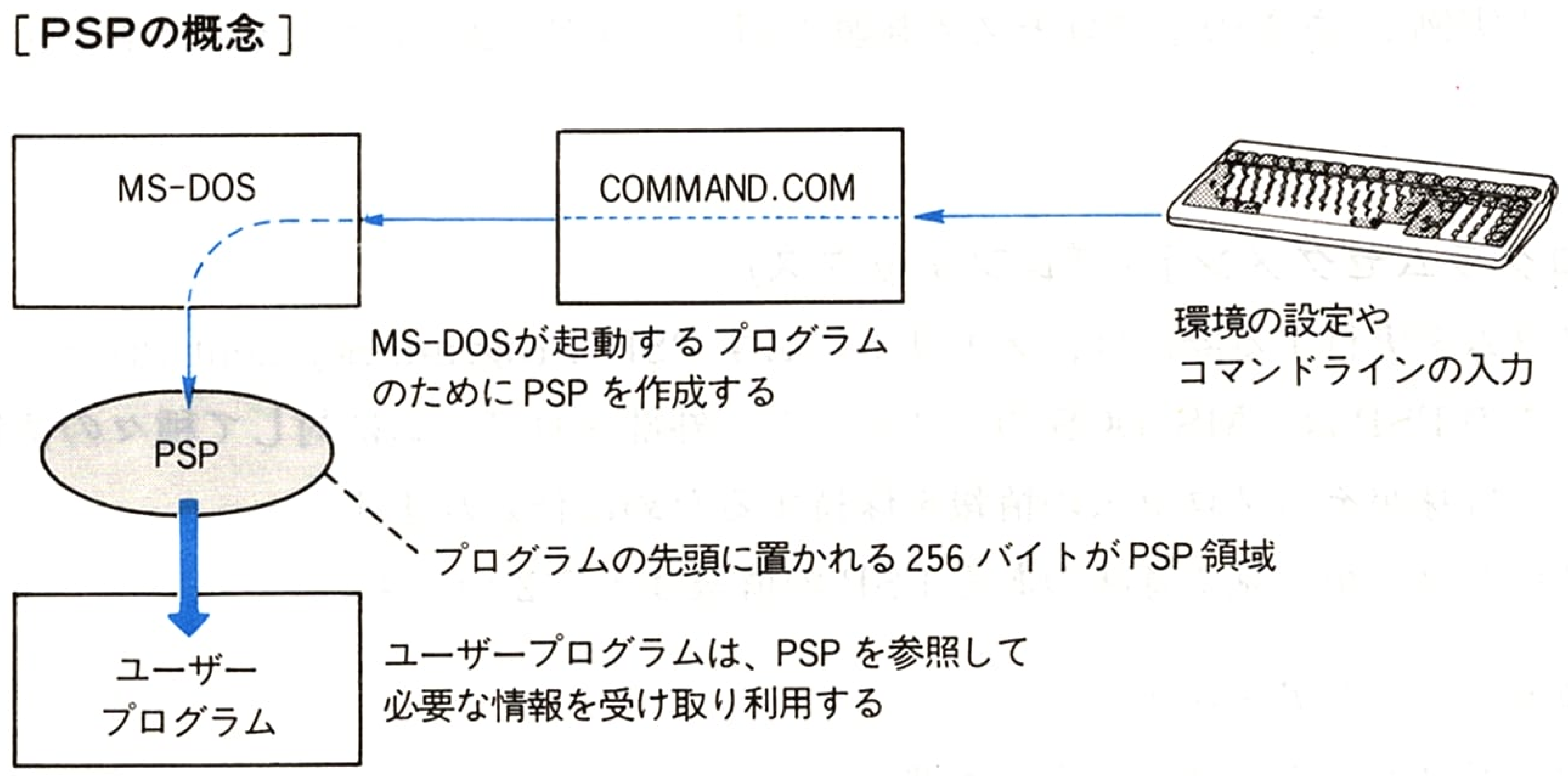
環境セグメントアドレスとは、COMMAND.COM の SET コマンドで設定した、環境文字列が格納されている領域の先頭アドレスのことであり、この領域には環境文字列が、たとえば、「PATH　A：¥；A：¥BIN」というような入力時そのままの形式で格納されています。必要があれば、ユーザープ

ログラムにおいて、このアドレスから環境文字列を取り出して利用することも可能です。

「コマンドラインのパラメータ文字列」には、ユーザーが入力したコマンドラインの文字列のうち、コマンド名およびリダイレクト、パイプ記号を除いた部分が格納されており、この領域の文字列を取り出すことにより、パラメータとして与えられたファイル名やほかの文字列などを、ユーザープログラムが受け取ることができます。

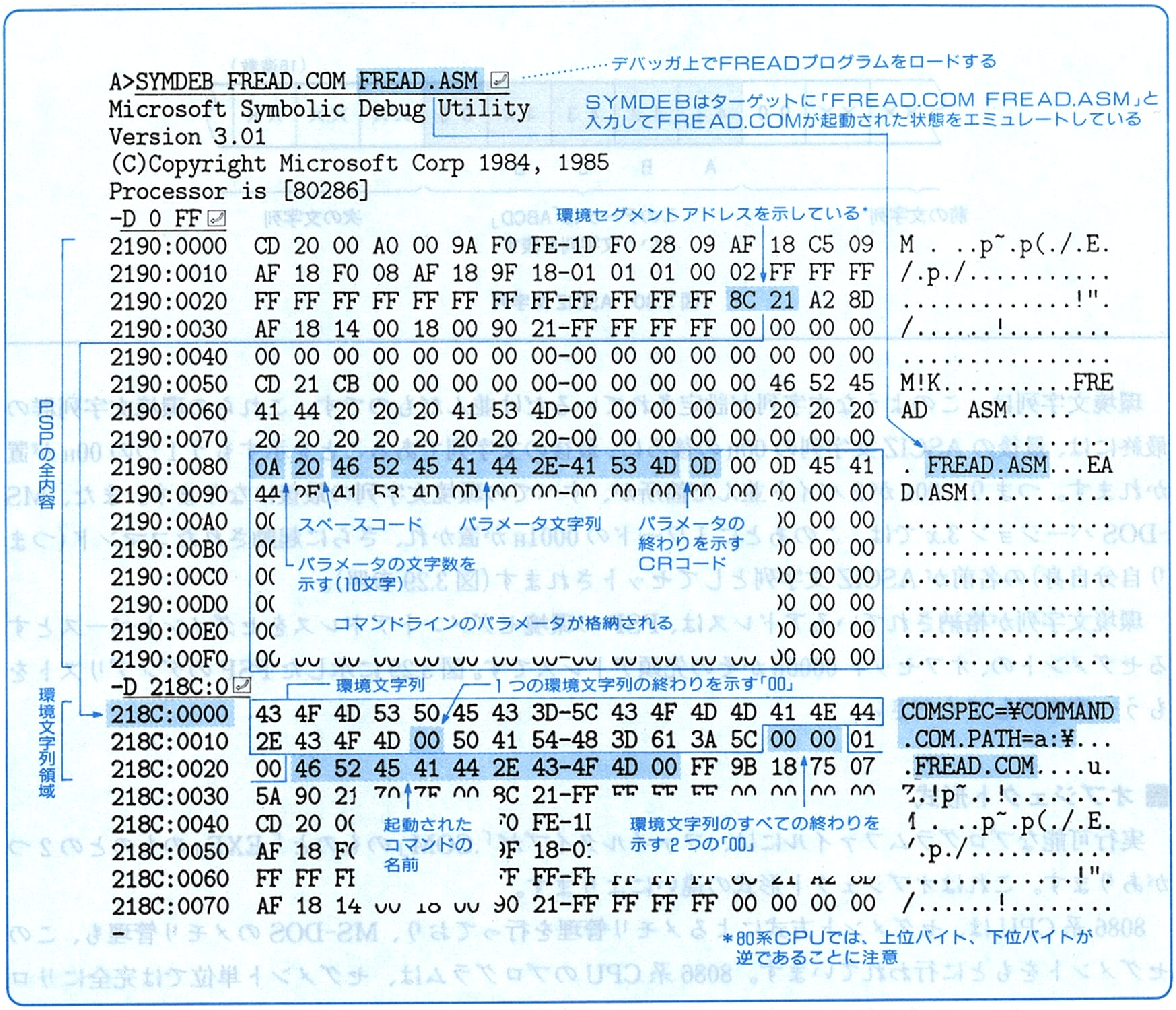
COMMAND.COMが外部プログラムを実行する際には、ユーザーが入力したコマンドラインの情報がMSDOS.SYSに渡され、これを受け取ったMSDOS.SYSはPSPを作成して、そこに各種の情報を格納します。外部プログラムは、このPSPを介してシステムの環境文字列やコマンドラインの文字列を受け取ることができるのです。つまり、外部プログラムから見ると、この2つの情報を受け取るためにPSPが存在しているといえます。MSDOS.SYSから見ると、PSPはこの2つ以外にもいくつかの情報を保持しており、また役割もあるのですが、これについては後述します。ただしその中にある2つのFCBは、MS-DOSバージョン1.25やCP／Mとの互換性のために存在するものであり、無視してもかまいません(ただし、子プロセスを起動する場合は操作する必要がある)。

さて、PSPが作成されると、外部プログラムはそのPSPのすぐ後ろにロードされ、実行に移されます。ここで、2章で作成したファイル読み出しプログラムFREADを、MS-DOSの標準デバッガSYMDEB上でロードして、PSPの姿を実際に見てみましょう(**図3.29**)。SYMDEBでも、COMMAND.COMと同様のプロセス管理により、目的のプログラム(SYMDEBでは**ターゲットプログラム**と呼ぶ)をロードすることができます。


[PSPの概念]
MS-DOS
COMMAND.COM
MS-DOSが起動するプログラム
のためにPSPを作成する
環境の設定や
コマンドラインの入力
PSP
プログラムの先頭に置かれる256バイトがPSP領域
ユーザー
プログラム
ユーザープログラムは、PSPを参照して
必要な情報を受け取り利用する

— 図3.29 —

(次ページに続く)



```
A>SYMDEB FREAD.COM FREAD.ASM
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-D 0 FF
2190:0000  CD 20 00 A0 00 9A F0 FE-1D F0 28 09 AF 18 C5 09  M . ..p~.p(./.E.
2190:0010  AF 18 F0 08 AF 18 9F 18-01 01 01 00 02 FF FF FF  /.p./...........
2190:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 8C 21 A2 8D  .............!".
2190:0030  AF 18 14 00 18 00 90 21-FF FF FF FF 00 00 00 00  /......!........
2190:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
2190:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 46 52 45  M!K.........FRE
2190:0060  41 44 20 20 20 41 53 4D-00 00 00 00 00 20 20 20  AD   ASM.....
2190:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00          ........
2190:0080  0A 20 46 52 45 41 44 2E-41 53 4D 0D 00 0D 45 41  . FREAD.ASM...EA
2190:0090  44 2F 41 53 4D 0D 00 00-00 00 00 00 00 00 00 00  D.ASM...........
2190:00A0  00                                      00 00 00  ................
2190:00B0  00                                      00 00 00  ................
2190:00C0  00                                      00 00 00  ................
2190:00D0  00                                      00 00 00  ................
2190:00E0  00                                      00 00 00  ................
2190:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-D 218C:0
218C:0000  43 4F 4D 53 50 45 43 3D-5C 43 4F 4D 4D 41 4E 44  COMSPEC=¥COMMAND
218C:0010  2E 43 4F 4D 00 50 41 54-48 3D 61 3A 5C 00 00 01  .COM.PATH=a:¥...
218C:0020  00 46 52 45 41 44 2E 43-4F 4D 00 FF 9B 18 75 07  .FREAD.COM....u.
218C:0030  5A 90 21 70 7F 00 8C 21-FF FF FF FF 00 00 00 00  Z.!p~..!........
218C:0040  CD 20 00             70 FE-1D                    M . ..p~.p(./.E.
218C:0050  AF 18 F0             9F 18-01                    /.p./...........
218C:0060  FF FF FF             FF FF-FF FF FF FF 8C 21 A2 8D  .............!".
218C:0070  AF 18 14 00 18 00 90 21-FF FF FF FF 00 00 00 00  /......!........
-
```

デバッガ上でFREADプログラムをロードする
SYMDEBはターゲットに「FREAD.COM FREAD.ASM」と入力してFREAD.COMが起動された状態をエミュレートしている
環境セグメントアドレスを示している*
PSPの全内容
スペースコード
パラメータ文字列
パラメータの終わりを示すCRコード
パラメータの文字数を示す(10文字)
コマンドラインのパラメータが格納される
環境文字列
1つの環境文字列の終わりを示す「00」
環境文字列領域
起動されたコマンドの名前
環境文字列のすべての終わりを示す2つの「00」
*80系CPUでは、上位バイト、下位バイトが逆であることに注意

図 3.29　FREAD プログラム実行時の PSP の内容を見る

■ 環境文字列の形式

環境文字列の具体的な形式は、ASCII コードで表された文字列の最後に、1 バイトの 00H が付け加えられた形をしています。これを ASCIZ 文字列（アスキーズィー文字列：ASCII＋ZERO の意味）と呼び、ファイル名（パス名）を表す場合など、内部的にはこの形式を用いています（図 3.30）。この ASCIZ による文字列の形式は、C 言語で使われる文字列の基本形式であり、MS-DOS と C 言語とのインターフェイスが有利になります。

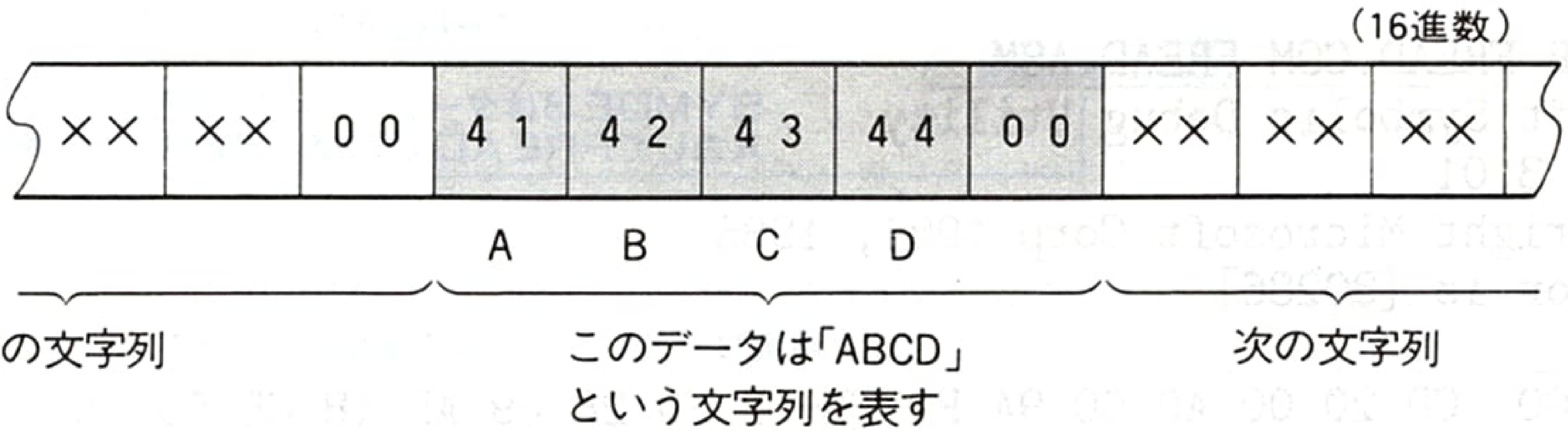


図3.30　ASCIZ文字列

環境文字列は、このような文字列が設定されているだけ並んだものです。これらの環境文字列群の最終には、最後のASCIZ文字列の00_Hの後ろに、最後の文字列であることを示すもう1つの00_Hが置かれます。つまり、00_Hが2バイト並んだ箇所が、すべての環境文字列の最後になります。また、MS-DOSバージョン3.*x*では、このあとに1ワードの0001_Hが置かれ、さらに起動されたコマンド(つまり自分自身)の名前がASCIZ文字列としてセットされます(図3.29参照)。

環境文字列が格納されているアドレスは、PSPの環境セグメントアドレスをセグメントベースとするセグメントの、オフセット0000_Hがその先頭アドレスです。図3.29に示したPSPのダンプリストをもう一度参照してください。

■ オブジェクト形式

実行可能なプログラムファイルには、ファイルタイプが「.COM」のものと「.EXE」のものとの2つがあります。これはオブジェクト形式の違いによります。

8086系CPUは、セグメント方式によるメモリ管理を行っており、MS-DOSのメモリ管理も、このセグメントをもとに行われています。8086系CPUのプログラムは、セグメント単位では完全にリロケータブル(再配置可能)です。セグメントレジスタを変更しなければ、プログラムをセグメントごとほかのアドレスへ移動しても、そのままの状態で実行が可能です。ただし、1つのセグメントは、最大64Kバイトの大きさしかとれません。つまり、この場合のプログラムサイズは、64Kバイト以内のものに制限されるのです。この制約の解決策として、2つのオブジェクト形式が生まれました。

COM形式

COM形式は、プログラム全体(つまり、コード+データ+スタックなどのすべての合計)のサイズが1つのセグメントに収まるもの(64Kバイト以内：8080やZ80などと同等)であり、これを実行するには、プログラムの実行が行われるセグメントだけをセグメントレジスタにセットしておけばよく、実行中にセグメントレジスタを変更する必要はありません。つまり、プログラムファイルとして存在するものを、そのままの形でロードして実行することができるのです。ですから、後述するEXE形式に

比べて、プログラムファイルのサイズが小さく、プログラムのロードから実行までに要する時間が少なくてすみます。ただし、プログラム全体のサイズが、64K バイト以内でなければなりません(動的にメモリをアロケートする場合は別として)。

COM 形式のプログラムの実行開始アドレスは、必ずオフセット 0100_{H} であり、オフセット 0000_{H} から $00FF_{\mathrm{H}}$ までの 256 バイトは、前述の PSP の領域としてシステムによって使われます(**図 3.31** 参照)。なお、すべてのセグメントレジスタは同じ値(この PSP の先頭アドレス)を持ち、論理セグメントを 1 つ持っているだけです。

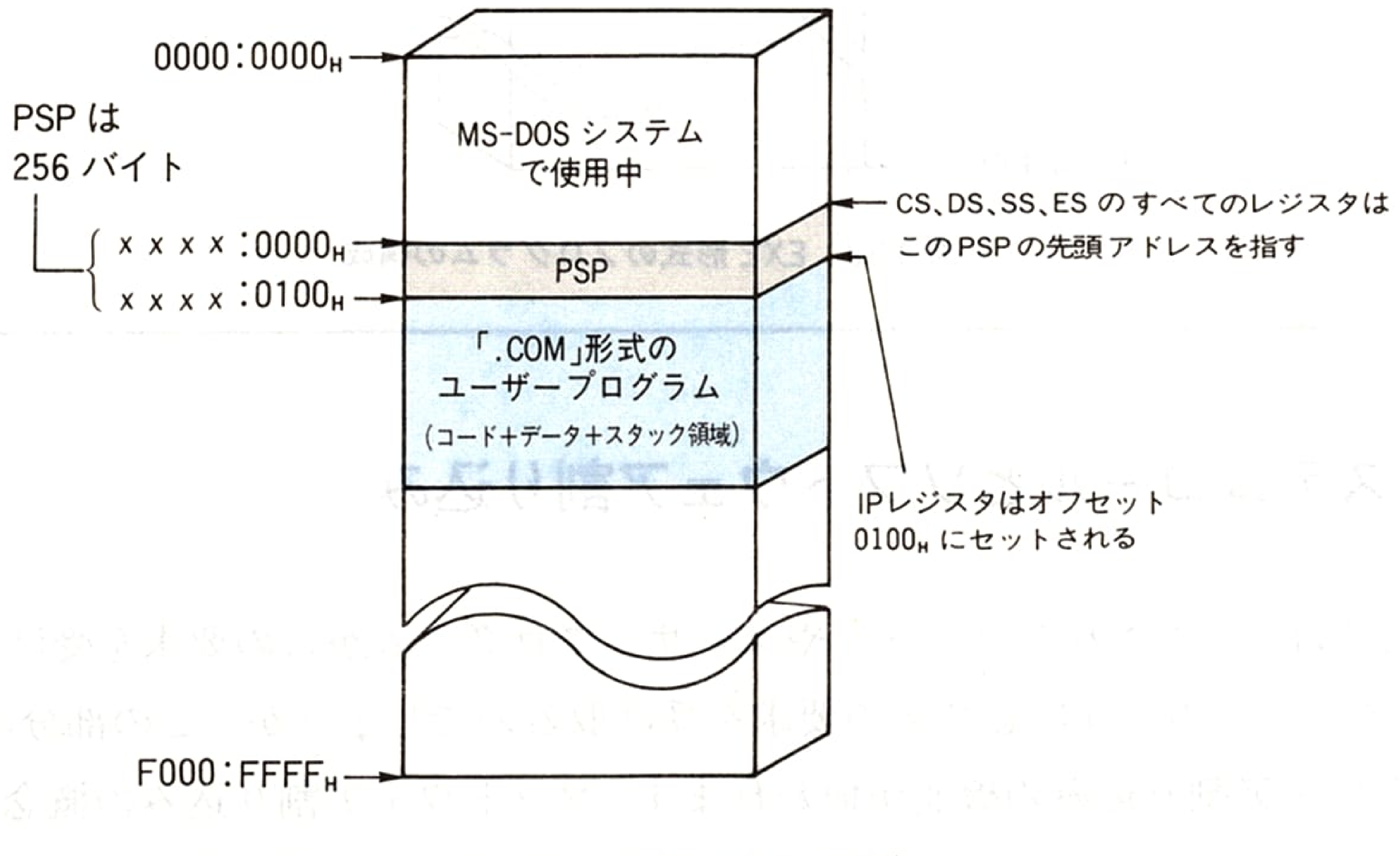

0000:0000H
PSP は
256 バイト
x x x x :0000H
x x x x :0100H
MS-DOS システム
で使用中
PSP
「.COM」形式の
ユーザープログラム
(コード+データ+スタック領域)
CS、DS、SS、ES のすべてのレジスタは
この PSP の先頭アドレスを指す
IPレジスタはオフセット
0100H にセットされる
F000:FFFFH

図 3.31　COM 形式のプログラムの構成

EXE 形式

EXE 形式は、COM 形式とは異なり、リロケート操作を必要とします。EXE 形式のプログラムは、MS-DOS システムの状態により、ロードされるアドレスが変化するからです。8086 系 CPU のプログラムは、セグメント単位でのみリロケータブルなので、論理セグメント値を参照する部分を、実際にロードされたアドレスに直さなければなりません。この処理がリロケート操作であり、EXE 形式のプログラムは、このリロケート情報をファイル中に持たなければならないため、プログラムのサイズが大きくなります。さらに、このプログラムのロード時には、リロケート操作が行われるため、ロードして実行するまでの時間は、COM 形式より長くかかります。しかし、プログラム全体のサイズが 64K バイトを超える場合は、この EXE 形式のほかに実現の方法はありません。

EXE 形式では、プログラムが起動した時点での CPU の DS、ES の各レジスタに PSP のセグメントがはいっており、そのオフセット 0000_{H} から $00FF_{\mathrm{H}}$ までが PSP の領域です(**図 3.32**)。

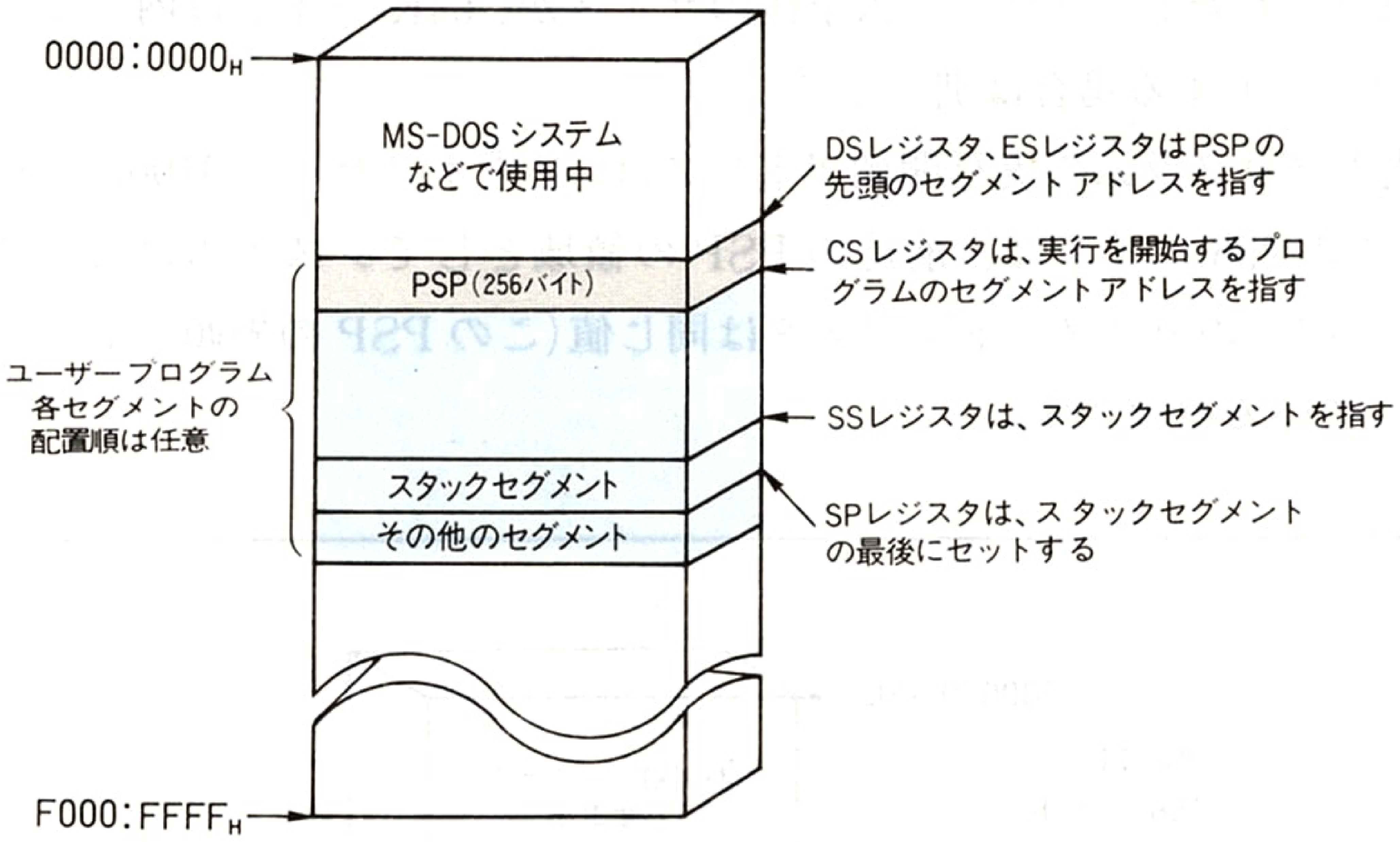

0000:0000H
MS-DOSシステムなどで使用中
PSP（256バイト）
ユーザープログラム各セグメントの配置順は任意
スタックセグメント
その他のセグメント
F000:FFFFH
DSレジスタ、ESレジスタはPSPの先頭のセグメントアドレスを指す
CSレジスタは、実行を開始するプログラムのセグメントアドレスを指す
SSレジスタは、スタックセグメントを指す
SPレジスタは、スタックセグメントの最後にセットする

図3.32　EXE形式のプログラムの構成

3.7.2　システムコールとソフトウェア割り込み

MSDOS.SYSは、入力されたコマンドやユーザープログラムからの要求を受け取り、それに応じた処理を行いますが、どのようにしてその要求を受け取るのでしょうか。この部分の処理には、8086系CPUの**ソフトウェア割り込み**の機能が使われます。ソフトウェア割り込みの概念は、ソフトウェア開発者にとって（高級言語を使うにしても）MS-DOSの処理の仕組みを知る上で重要な知識の1つです。

割り込み（インタラプト）の本来の機能は、CPUによるプログラムの実行とは無関係、非同期に発生する外部の要因による入出力要求などを処理するためのものです。具体的には、まず外部の要因が発生すると、そこからの信号がハードウェア的にCPUに与えられ、CPUはその信号が入力されたのを「きっかけ」として、現在実行している仕事を一時中断し、外部が要求する仕事を行い（そのプログラムはあらかじめ用意されている）、その実行が終了すると、中断していたもとの仕事を再開します。文字どおり、実行中の処理に割り込んで別の処理を行うのです。

この割り込みには、複数の処理の中から任意の1つを選択して実行する機能がありますが、その処理を選択する方法はCPUによって異なります。一般には、メモリ上のある決まった場所に、割り込みにより実行される各種のプログラムの実行開始アドレスを示すテーブルを置き、割り込みが発生すると、そのテーブル上の目的のアドレスへ制御を移すという方法がとられます。8086系CPUもこの方式であり、メインメモリの先頭（最下位アドレス）の1Kバイトがこのテーブルのために使われます。これを**割り込みベクタテーブル**と呼んでいます。

さて、割り込みの要求は、本来前述のような外部の要因によって、そこからハードウェア的に与えられるもので、このような割り込みを**ハードウェア割り込み**あるいは**外部割り込み**と呼んでいます。一方この割り込み要求を、CPU の命令として用意し、内部のプログラムからソフトウェア的に割り込みをかける機能があり、この割り込みを**ソフトウェア割り込み**あるいは**内部割り込み**と呼んでいます（図 3.33 参照）。

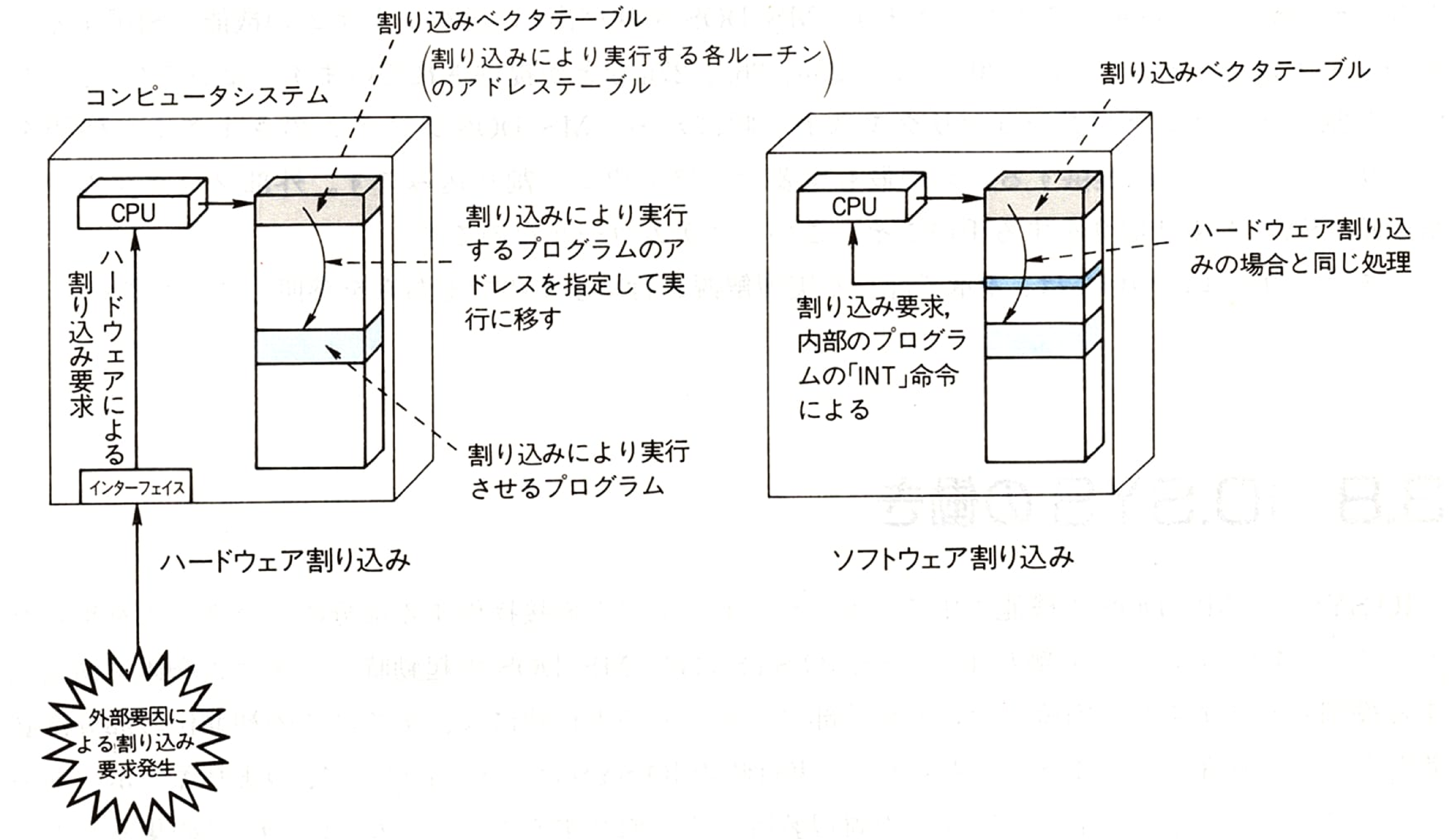

割り込みベクタテーブル
（割り込みにより実行する各ルーチンのアドレステーブル）
コンピュータシステム
CPU
ハードウェアによる割り込み要求
インターフェイス
割り込みにより実行するプログラムのアドレスを指定して実行に移す
割り込みにより実行させるプログラム
ハードウェア割り込み
外部要因による割り込み要求発生
割り込みベクタテーブル
CPU
ハードウェア割り込みの場合と同じ処理
割り込み要求，内部のプログラムの「INT」命令による
ソフトウェア割り込み

図 3.33　ハードウェア割り込み、ソフトウェア割り込みの概念

ソフトウェア割り込み命令は、プログラムの分岐という面からは、サブルーチンコール命令と変わりはありませんが、目的のルーチンの呼び出し方に大きな違いがあり、そこにソフトウェア割り込み命令を使うメリットがあります。サブルーチンコール命令の場合は、コールする際に、サブルーチンが置かれているメモリ上のアドレスを指定しなければなりませんが、各種のサブルーチンのそれぞれのアドレスは、機種によって異なる可能性があり、繁雑であるばかりか、ユーザープログラムの共通性を保てません。

ところがソフトウェア割り込み命令の場合は、割り込みテーブルにあらかじめ各種処理ルーチンのアドレスを機種ごとにセットして、それを公表しておけば、サブルーチンコールの場合のように、システム内の各種ルーチンのそれぞれのアドレスを知る必要がなくなります。つまりユーザープログラ

ムは、各機種間で統一された割り込みテーブルを利用することにより、システムから独立することができるわけで、ソフトウェア割り込みを使ってシステムを呼び出す限りにおいては、どのシステム上でも動作する共通のプログラムを作ることができます。

8086系CPUの割り込みは、00_H～FF_Hの番号を付けた256種類のルーチンを選択する機能があります。このうち、ソフトウェア割り込み用には、20_H～FF_Hまでが割り当てられており、MS-DOSシステムでは、この中の20_H～$3F_H$の32個を使用しています。

ユーザープログラムからは、このソフトウェア割り込みを介して、MS-DOSに種々の処理を要求したり、その結果を受け取ることができます。MS-DOSを呼び出し、そのシステムの機能を利用するためのシステムコールとしては、20_H、21_H、25_H、26_H、27_Hなどが提供されています。なかでも、21_Hを使った割り込みは**ファンクションリクエスト**と呼ばれる、MS-DOSシステムのさまざまな機能をユーザープログラムに提供するための最も重要なソフトウェア割り込みです。外部プログラムが、MSDOS.SYSとやり取りをする手段こそ、このシステムコールなのです。

システムコールについては、4章で詳しく実習解説を行いますのでそちらを参照してください。

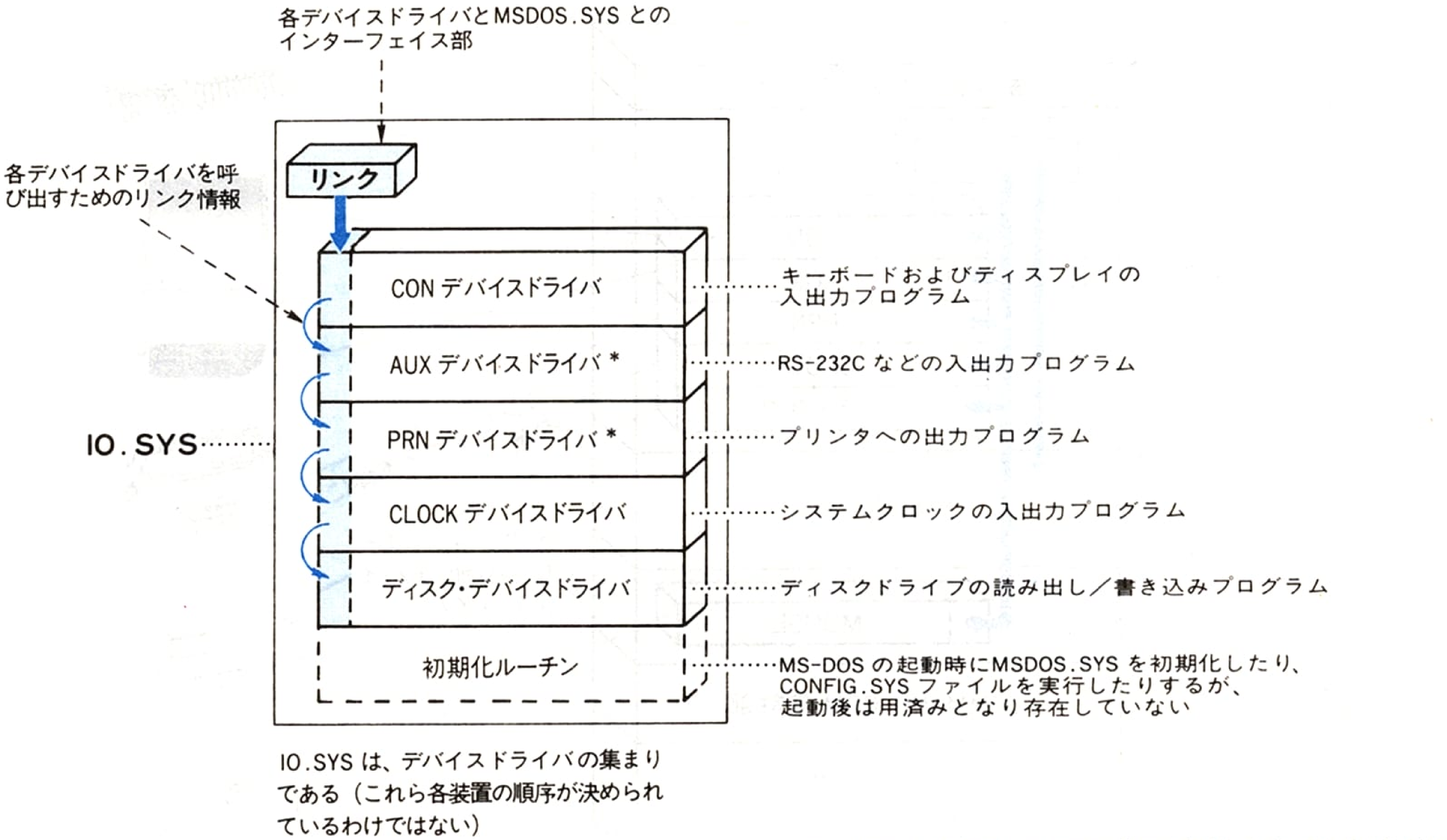
3.8　IO.SYSの働き

IO.SYSは、MS-DOSの機能の中で、主にハードウェアを直接操作する部分のプログラムを集めたものです。本章の3.2.2でも触れましたが、IO.SYSには、MS-DOSの起動時にシステム全体を初期化する役割もありますが、内蔵コマンドや外部プログラムの実行時には、すでにこの初期化の部分は必要がないため存在していません。プログラム実行時のIO.SYSは、ハードウェア、つまりキーボードやディスプレイ、ディスクドライブなどの周辺装置とやり取りするモジュール(ルーチン)の集まりとして存在しています。この各モジュールが、**デバイスドライバ**と呼ばれるものです。

3.8.1　デバイスドライバ

各種周辺装置の直接の制御プログラムはコンピュータの機種によって異なりますが、MS-DOSがそれらの周辺装置を利用するときは、各装置ごとに用意された管理モジュールを介して、決められた方法で行われます。この管理モジュールがデバイスドライバであり、これを呼び出す方法や、コマンドを実行する際に行う処理などが統一されています。このような周辺装置の管理モジュールであるデバイスドライバは、**図3.34**のような構成になっています。

IO.SYSの大半は、このようなデバイスドライバであり、実行時にはデバイスドライバの集まりとして扱われます。


各デバイスドライバとMSDOS.SYSとの
インターフェイス部
各デバイスドライバを呼
び出すためのリンク情報
リンク
IO.SYS
CONデバイスドライバ
AUXデバイスドライバ *
PRNデバイスドライバ *
CLOCKデバイスドライバ
ディスク・デバイスドライバ
初期化ルーチン
キーボードおよびディスプレイの
入出力プログラム
RS-232Cなどの入出力プログラム
プリンタへの出力プログラム
システムクロックの入出力プログラム
ディスクドライブの読み出し／書き込みプログラム
MS-DOSの起動時にMSDOS.SYSを初期化したり、
CONFIG.SYSファイルを実行したりするが、
起動後は用済みとなり存在していない
IO.SYSは、デバイスドライバの集まり
である（これら各装置の順序が決められ
ているわけではない）

＊バージョン3.1以降のPC-9800シリーズ用MS-DOSではAUXおよび
PRNデバイスドライバはIO.SYSに含まれていないものもある。

図 3.34　IO.SYS の構成

デバイスドライバは、普通、1つの周辺装置に1つ用意されますが、ディスクドライブなどでは、複数のドライブや種類の異なるドライブを、1つのデバイスドライバで管理するのが普通です。コンソールのデバイスドライバも、実際には、入力はキーボード、出力はディスプレイという別々の装置を扱うわけですが、これも普通は、1つのデバイスドライバでその2種類の装置を扱っています。一般に、ミニコン以上のコンピュータでは、コンソールといえばターミナル装置のことを意味し、入力も出力も同じ1つの装置「ターミナル」が対象なのです。一般のパーソナル・コンピュータでは、そのターミナルの機能をコンピュータ本体が持っている（キーボードおよびディスプレイが付属している）ため、パーソナル・コンピュータから見れば、入力／出力が別々の装置のように考えられますが、本来は1つのターミナル装置なのです。場合によっては、コンソールとして実際にターミナル装置を外部接続しなければならないパーソナル・コンピュータもあるかもしれません。要するに、論理的に区別できる装置の1つひとつに対して、それぞれデバイスドライバが用意されるのです。（図 3.35 参照）

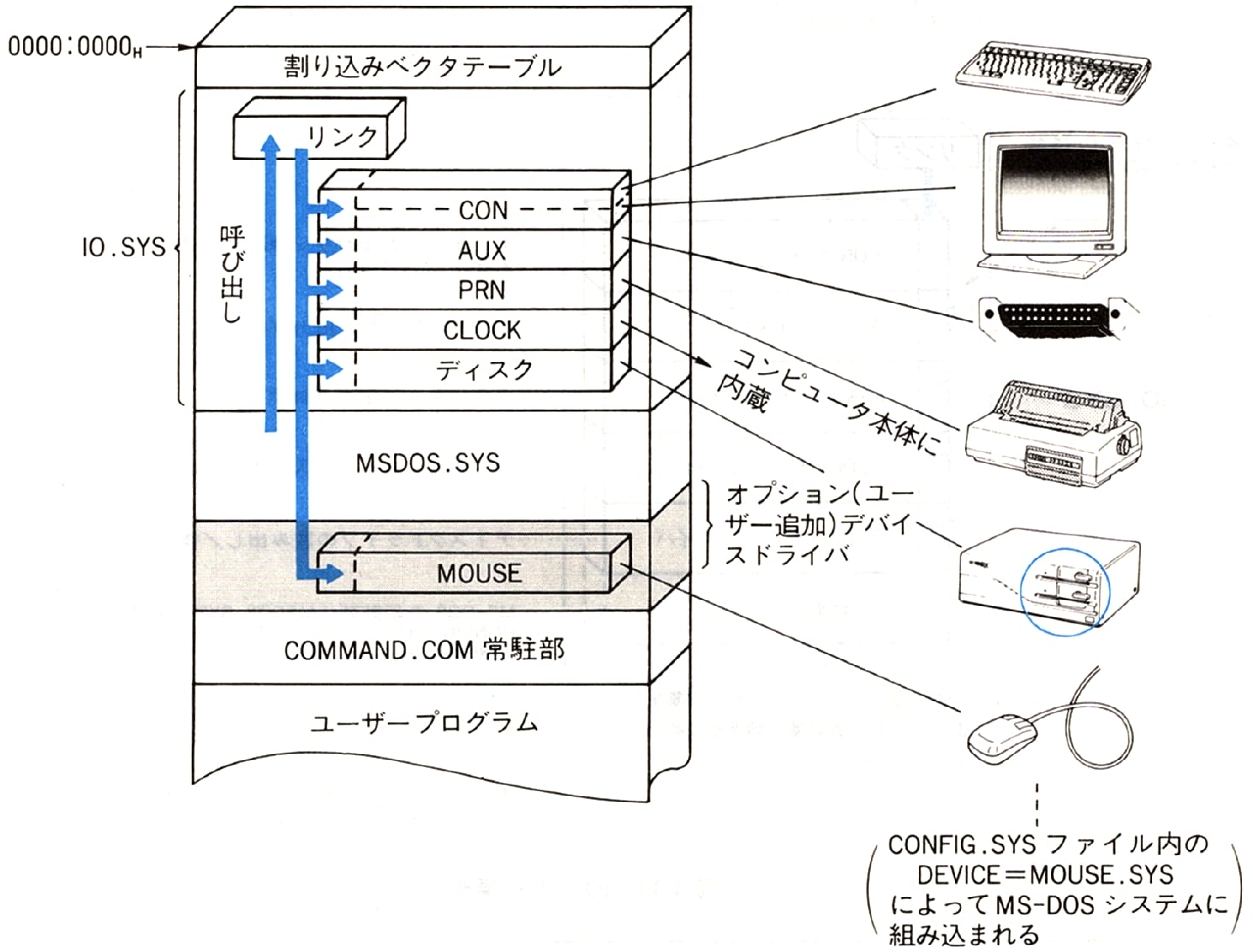

0000:0000H
割り込みベクタテーブル
IO.SYS
リンク
呼び出し
CON
AUX
PRN
CLOCK
ディスク
コンピュータ本体に内蔵
MSDOS.SYS
オプション(ユーザー追加)デバイスドライバ
MOUSE
COMMAND.COM 常駐部
ユーザープログラム
CONFIG.SYS ファイル内の DEVICE＝MOUSE.SYS によってMS-DOS システムに組み込まれる

図3.35　MS-DOSとデバイスドライバの関係

3.8.2　ブロック型デバイスとキャラクタ型デバイス

MS-DOSで取り扱うデバイス(周辺装置)は、入出力の形式の違いから、ブロック型デバイスとキャラクタ型デバイスとに分けられています。

ブロック型とは、ディスクドライブのように、一度にまとまった量のデータを、ランダムアクセスできるようなデバイスのことをいいます。つまり、ある決まった量を1ブロックとするデータが、複数ブロック存在し、そのブロック単位で任意のブロックをアクセスすることが可能なデバイスのことです。

それに対してキャラクタ型デバイスとは、コンソールやプリンタなどのように、一度に1バイトずつのデータをやり取とりするデバイスのことで、通常、ランダムアクセスはできません(図3.36)。

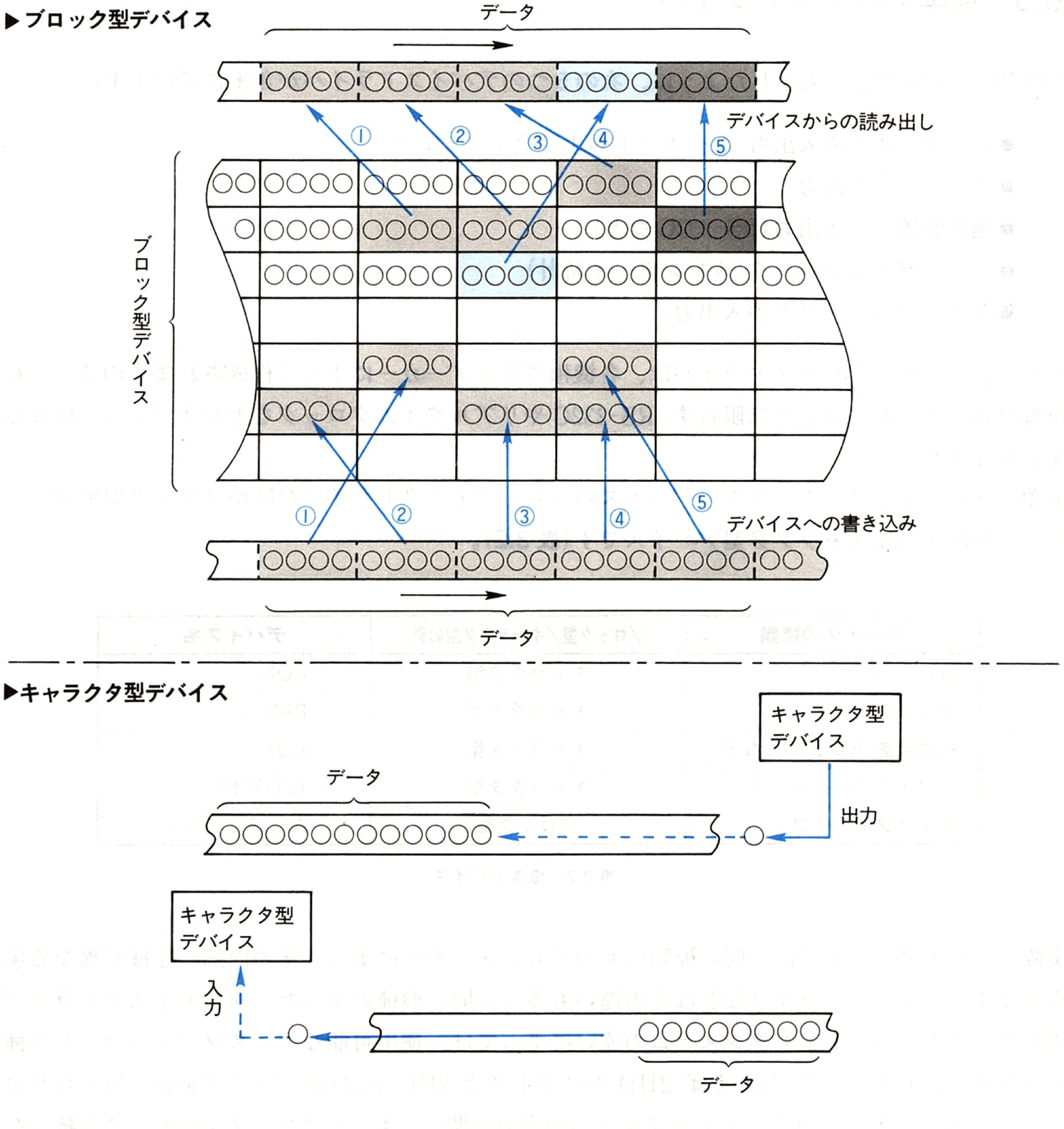

▶ブロック型デバイス
データ
①
②
③
④
⑤
デバイスからの読み出し
ブロック型デバイス
①
②
③
④
⑤
デバイスへの書き込み
データ
▶キャラクタ型デバイス
キャラクタ型デバイス
出力
データ
キャラクタ型デバイス
入力
データ

図 3.36　ブロック型デバイス、キャラクタ型デバイスの構造

3.8.3　標準デバイスドライバ

IO.SYSには図3.35にも示したように、次の5つのデバイスドライバが含まれています。

- ●コンソールとの入出力（キーボードおよびディスプレイ）
- ●プリンタへの出力
- ●補助装置との入出力（RS-232Cなど）
- ●リアルタイムクロック（カレンダー＋時計）
- ●ディスクドライブとの入出力

ただし、これらのデバイスドライバは、各機種ごとにメーカーによって作成されますので、これらが全部サポートされているとは限らず、RS-232Cやリアルタイムクロックなどがオプションになる機種もあります*。

前項で述べたように、この5つのデバイスのうち、ディスクドライブだけがブロック型デバイスであり、ほかはすべてキャラクタ型デバイスです（**表3.2**）。

デバイスの種類	ブロック型／キャラクタ型の別	デバイス名
コンソール	キャラクタ型	CON
プリンタ	キャラクタ型	PRN
補助装置（RS-232Cなど）	キャラクタ型	AUX
リアルタイムクロック	キャラクタ型	(CLOCK)
ディスクドライブ	ブロック型	A:、B:、C:……

表3.2　標準デバイス

実際のデバイスドライバは、同じ種類のものでも、メーカーによってその内部の仕様が異なる場合があります。ディスクドライブなどはその違いも多く、同一機種のまったく同じディスクドライブを使用していても、コンピュータシステムの違いによっては、使用可能なディスクフォーマットの種類に差があることもあります（たとえば、2HDのドライブで、2DDや2Dのディスクを読み出したり書き込んだりすることなど）。また、ディスプレイへの出力に関しても、エスケープシーケンスの統一などがとれておらず、たとえば、文字のリバースやブリンクなどに対する機能やその属性もまちまちです。

＊ バージョン3.1以降のPC-9800シリーズ用MS-DOSでは、AUXデバイスおよびPRNデバイスはオプションとなり、CONFIG.SYSファイルに登録しておかなければ利用できないものもある。

しかし、このような違いは、MSDOS.SYS の関知しないそれぞれの機種独自のことであり、デバイスドライバとしての決められた手順に従った入出力がサポートされていれば、MS-DOS を動かすことができます。ただ、それぞれの装置独自のハードウェアの性能を、どこまで有効に引き出すかについては、各メーカーがどこまで熱心にデバイスドライバを作成するか(つまりは、MS-DOS をサポートするか)にかかっています。

それぞれの周辺装置は、各メーカー独自のハードウェアで構成されているため、同じ種類の周辺装置でも、当然その直接の制御方法は異なることになります。そこで各周辺装置ごとにデバイスドライバを作成することにより、機種による制御方法の違いなどが吸収され、MSDOS.SYS から見ると、どの機種のハードウェアでも、周辺装置の種類別に統一された取り扱いができることになるのです。デバイスドライバの概念で重要なのはこのことなのです。

3.8.4　デバイスファイル(キャラクタ型デバイス)

キャラクタ型デバイスは、それぞれの装置名を持っており、その名前によって識別されます。**デバイスファイル**[*]と呼ばれるこれらには、次のものがあります。

CON………コンソール
AUX………補助入出力装置(たとえば RS-232C インターフェイス)
PRN………プリンタ

MS-DOS では、周辺装置もファイルとみなし、その扱いもファイルに対する入出力と同等に行います。つまり、入出力の対象となるものをすべてファイルとみなすのです。たとえば、日常よく使うコマンドライン

```
A>COPY  CON  AUTOEXEC.BAT⏎
```

によって、コンソール(キーボード)から入力した内容のテキストファイルが、簡単に作成できることはよくご存知でしょう。

2 章のファイルシステムのところでは触れませんでしたが、ファイルをオープンする際には、入力されたファイル名によるディレクトリの検索を行う前に、それがデバイスファイルかどうかのチェックが行われます。もちろん、それがデバイスファイルであれば、ディスク上のファイルではなく、その

---

[*] デバイスファイルとして扱えるものには、前述の標準デバイスのほかに、NUL デバイスがある。これは、ダミーの入出力のために使われるファイルで、出力されるデータを捨てるためなどに用いられる。この NUL デバイスに対するデバイスドライバは MSDOS.SYS 内にあり、呼ばれると何もせずに戻るだけの働きをする。

装置のデバイスドライバを相手に入出力を行うことになります。このため、さきにあげたデバイス名と同じ名前のファイルを作ることはできないのです。

のちほど解説しますが、CONFIG.SYSファイルによって、ユーザー独自のキャラクタ型のデバイスドライバを登録した場合も、その装置にデバイス名を付けることができ、デバイスファイルとして同様に取り扱うことができます*。また、登録する際に、ユーザーデバイス名を、標準デバイスと同じ名前(CONやPRNなど)にすると、標準のデバイスドライバが無効となり、ユーザー・デバイスドライバが新たにCONデバイスやPRNデバイスなどとして使われるようになります。

3.8.5　ドライブ(ブロック型デバイス)

ブロック型デバイスは、「A：」「B：」「C：」…、というドライブ名で区別されます。複数のドライブを1つのデバイスドライバでサポートすることもでき、この場合は、1つのデバイスドライバが複数のドライブ名を持つことになります(これが普通)。たとえば、図3.37のようになります。

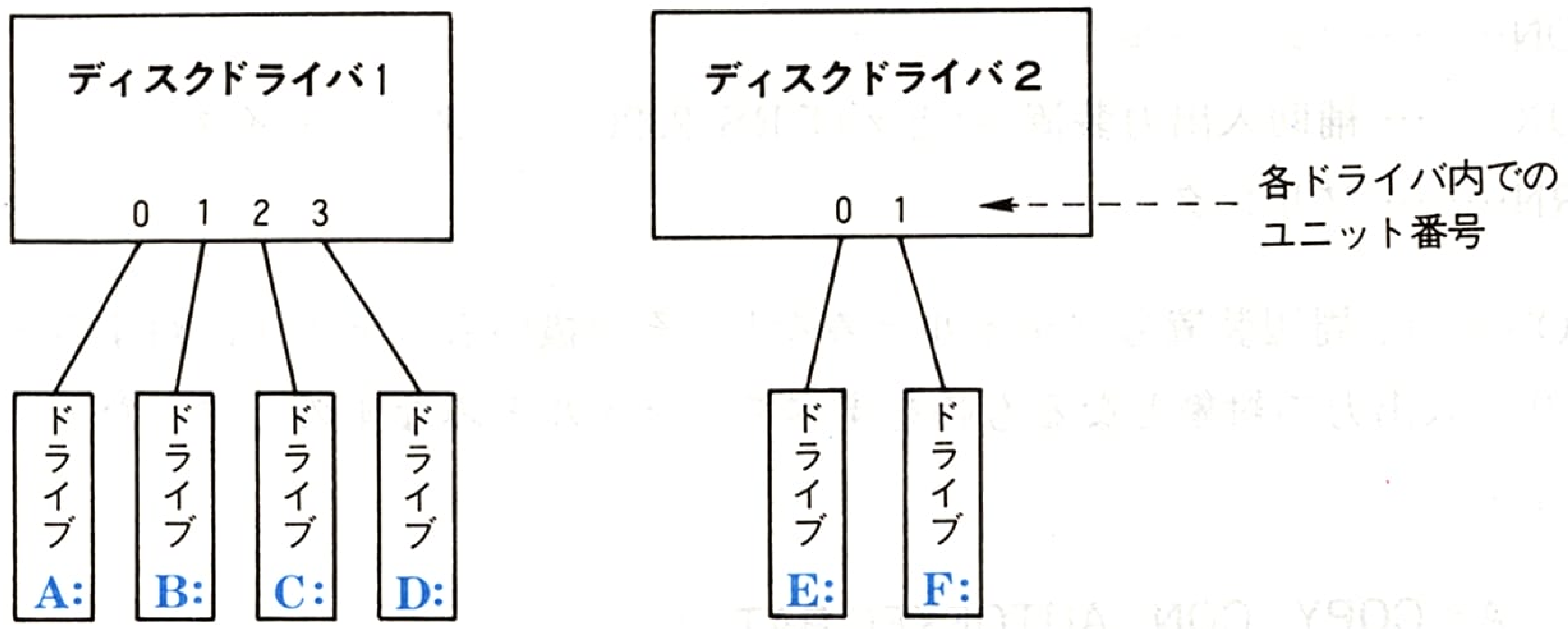

ディスクドライバ1
0 1 2 3
ディスクドライバ2
0 1
各ドライバ内での
ユニット番号
ドライブ A:
ドライブ B:
ドライブ C:
ドライブ D:
ドライブ E:
ドライブ F:

図3.37　ディスクドライブとそのデバイスドライバ

のちほど解説しますが、CONFIG.SYSファイルによって、ブロック型のデバイスドライバを登録すると、そのデバイスドライバのサポートするディスクは、現在のデバイスドライバの後ろに追加されるかたちとなり、標準のドライブに続くドライブ名を持つことになります(図3.38参照)。

＊ 7章で「PLT」という名前のデバイスドライバを作成している。

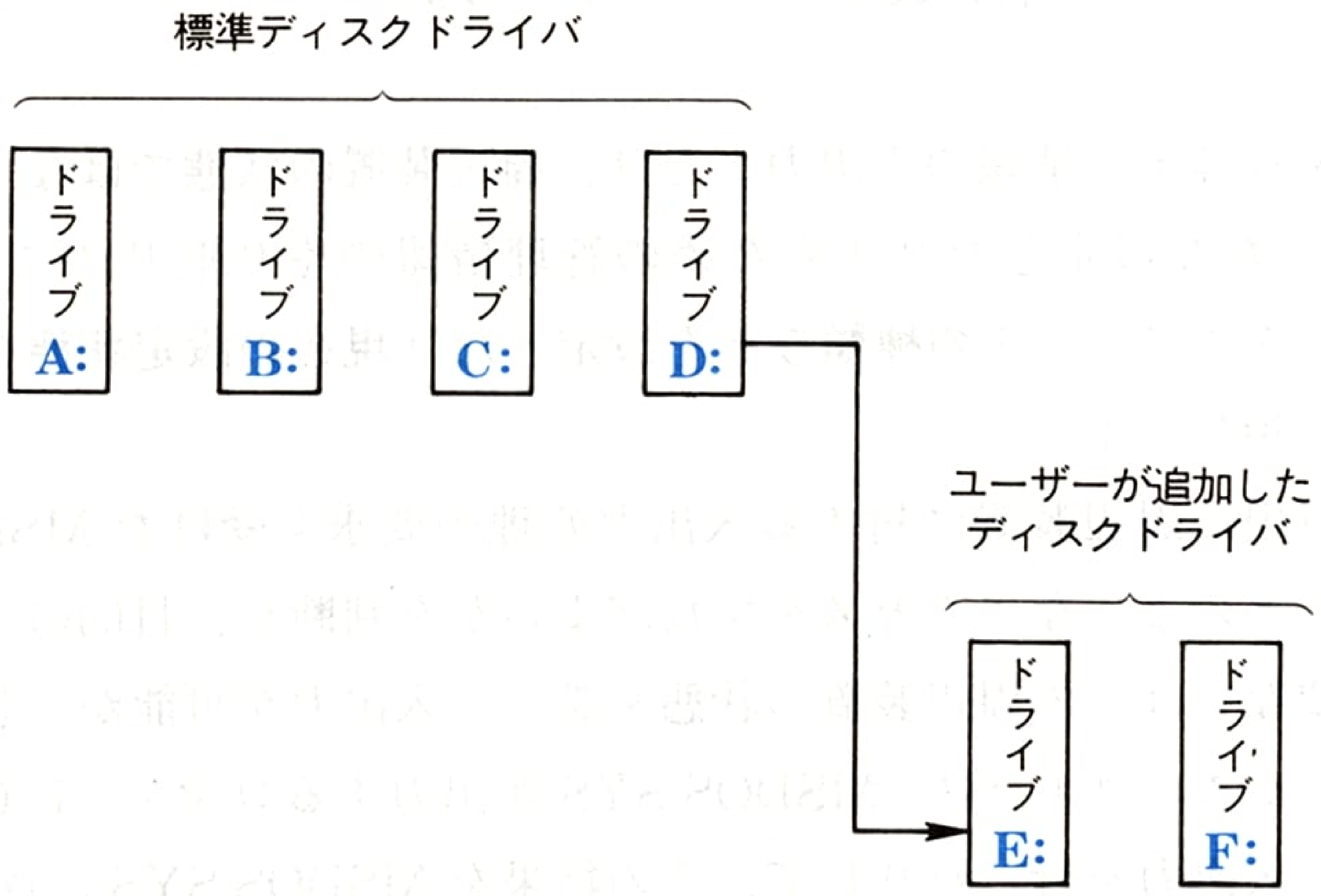


図 3.38　新しいブロック型デバイスドライバの追加

ブロック型デバイスにおいては、データの入出力やその管理に、2 章で解説したファイルシステムが構成されます。このブロック型デバイスのハードウェアは、何もディスクドライブだけとは限りません。ランダムアクセスの入出力ができれば何でもよいわけで、たとえばメモリ(RAM)でもよいのです。このようなディスク以外の装置でも、デバイスドライバを介することにより、ディスクドライブと同様な取り扱いをすることができます。最近広く使われるようになった RAM ディスクと呼ばれるものが正にこれであり、RAM とのデータ転送を行うためのデバイスドライバを登録するだけで、RAM 上にディスクドライブと同じファイルシステムを構成することができるのです。

3.8.6　デバイスドライバとの入出力

各種の周辺装置とのデータの入出力は、実際にはデバイスドライバとのやり取りとして行われますが、具体的にどのようなやり取りが行われるのかを簡単に見ていきましょう。

デバイスドライバの処理は、ハードウェアが異なる各メーカーの装置を統一的に取り扱わなければならないため、たいへん複雑なものになっています。デバイスドライバの処理を大きく分けると次のようになります。

- 初期化
- 周辺装置の状態情報(ステータス)の取得
- 周辺装置とのデータの入出力
- デバイスドライバを管理する情報の入出力

「初期化」とは、MS-DOSの起動時に行われるシステムの初期化において、デバイスドライバがシステムに登録される際に、1度だけ行われるハードウェアおよびデバイスドライバ自身の初期化のことです。

「デバイスドライバを管理する情報の入出力」とは、周辺装置の状態ではなく、デバイスドライバそのものの状態を把握したり設定したりするための管理情報のやり取りのことで、たとえば、RS-232Cの各種のパラメータやプリンタの種類などを設定したり現在の設定状態を把握したりするための情報の入出力のことを指します。

外部プログラムの実行中に周辺装置に対する入出力処理の要求を受けたMSDOS.SYSは、どのデバイスドライバに対してどのようなアクセスをすればよいかを判断し、目的のデバイスドライバにコマンドを送ります。必要ならばその周辺装置の状態を調べ、入出力が可能か、状態が変化したかなどをチェックします。デバイスドライバは、MSDOS.SYSが出力するコマンドに従って、周辺装置の状態を調べたり、データの入出力を行ったりして、その結果をMSDOS.SYSに返します。

3.8.7　BPBとメディアチェック

デバイスドライバへのコマンドの与え方や結果の返し方などは、本章ではあまり触れないことにしますが(7章で、実際のデバイスドライバを作成する際に解説する)、ファイルシステムが構成されるブロック型デバイスドライバの、装置の状態についてだけ簡単に触れておきましょう。

MS-DOSでは、ランダムアクセスできるものであれば、何でも「ディスク」として使用することができ、それらはすべてブロック型デバイスとして、ファイルシステムの管理下で取り扱われます。通常のディスクドライブの種類には、3.5インチ、5インチ、8インチのフロッピーディスクやハードディスクなどがあり、さらに、それらに各種のフォーマットが存在します。MS-DOSでは、これらのすべてをディスクドライブとして使用することができ、しかも1つのディスクドライブで、メディアやフォーマットが異なるいくつかの種類のディスクをアクセスすることも可能です(たとえば、8インチの片面単密度と両面倍密度とか、5インチの2HDと2DDと2Dなど)。このようなことを可能にするのがこれから述べるBPBとメディアチェックの仕組みです。

ファイルシステムを構成する仕事――たとえば、ディレクトリやFAT、それにデータ領域などを作成してファイルを管理するのは、MSDOS.SYSの仕事です。デバイスドライバは、そういう仕事はまったく知る必要はありません。しかし、MSDOS.SYSは、現在どのドライブにどんな種類のディスクがセットされているかを常に把握しておかなければならず、このためデバイスドライバは、現在セットされているディスクの種類のチェックを行わなければなりません。この作業を**メディアチェック**と呼びます。

MSDOS.SYSがディスクを管理するには、たとえばそのディスクのディレクトリ領域やFAT領域の大きさがいくらであるかとか、1セクタのバイト数が何バイトであるかを知らなければなりません。

BPB(Bios Parameter Block)はこのような必要から考えられたもので、ひとことでいえば、各種ディスクのカタログです。これが人間であれば、さしずめ、各人の生年月日、身長、体重などといったデータでしょうか。このBPBは、ディスクを管理するために必要な情報が、ひとそろいディスクの種類ごとに書かれている表で、デバイスドライバ内に用意されています。

2.3.3で、標準ディスクフォーマットについて述べましたが、そのなかで、FATの最初の1バイトがFAT IDだったことを覚えているでしょうか。MS-DOS標準フォーマットのディスク(表2.1参照)であれば、このFATの先頭の1バイトを見ればディスクの種類を識別できます。つまり、デバイスドライバは、現在使用中のディスクのFAT IDの先頭バイトを参照し、その値(つまりディスクの種類)に対応するBPBを前述の表から取り出して、MSDOS.SYSに渡します。MSDOS.SYSは、与えられたこのBPB情報をもとに、ディスク管理、ファイル管理を行うのです(**図3.39** 参照)。

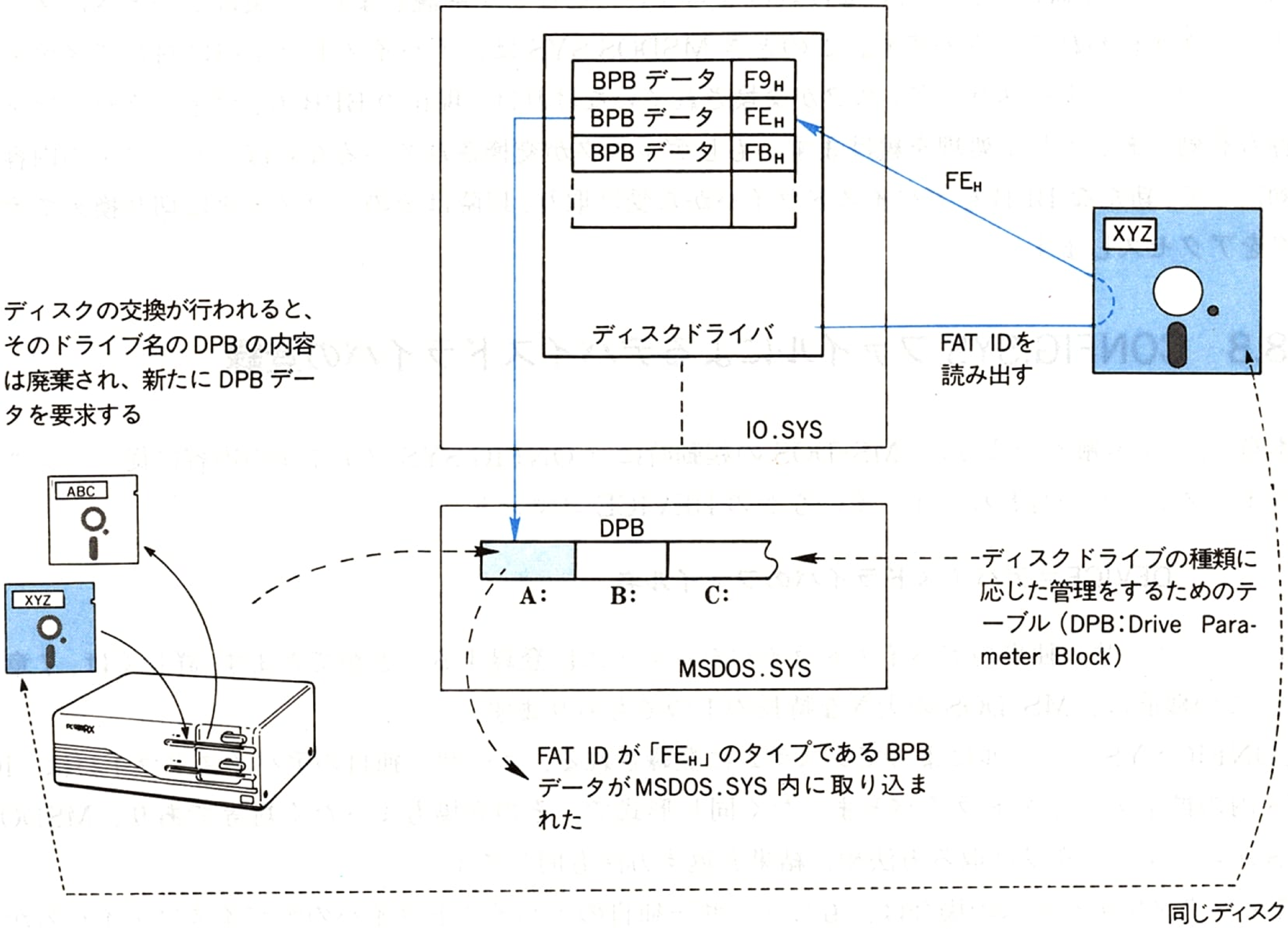

BPB データ　F9H
BPB データ　FEH
BPB データ　FBH
FEH
XYZ
ディスクドライバ
FAT IDを読み出す
IO.SYS
ディスクの交換が行われると、そのドライブ名のDPBの内容は廃棄され、新たにDPBデータを要求する
ABC
XYZ
DPB
A:
B:
C:
ディスクドライブの種類に応じた管理をするためのテーブル（DPB：Drive Parameter Block）
MSDOS.SYS
FAT IDが「FEH」のタイプであるBPBデータがMSDOS.SYS内に取り込まれた
同じディスク

図3.39　BPBとメディアチェックの概要

図3.39からも推測できるように、BPBの内容には、ディスクの種類を識別するための、5インチとか3.5インチといった物理的な形態の区別はなく、単にセクタのサイズや、ディレクトリの数や、FATが占めるセクタ数などが示されているだけです。このことからも、MS-DOSではランダムアクセスできる装置であれば何でも「ディスク」として使用できることがわかります。

さて、1つのドライブで異なる種類のディスクを使用するためには、MSDOS.SYSは常に現在セットされているディスクの種類を把握していなければなりません。そのためにデバイスドライバは、ディスクが取り換えられるたびに、そのことをMSDOS.SYSに知らせ、新しいディスクのBPBを渡す必要があります。このように、ディスクが交換されたかどうかを調べる動作がメディアチェックです。

ではメディアチェックはいつ行われるのでしょうか。

2章のファイルシステムについての解説で、ディスクのバッファリングについて触れました。ディスク・バッファリングは、バッファリングしたあとにディスクが交換されてしまっては、その動作が成立しません。このため、バッファの内容が有効かどうか、つまりディスクが交換されたかどうかを、MSDOS.SYSが正確に把握していなければならないことなどを解説しました。実はこのときにメディアチェックが行われているのです。このときMSDOS.SYSは、デバイスドライバに対してメディアチェックのコマンドを送り、ディスクが交換されていなければ、現在のBPBも、ディスクバッファの内容も有効であるとして処理を続けます。もしディスクが交換されているならば、バッファの内容を無効にして、新たなBPBをデバイスドライバから受け取り、以降はそのパラメータに切り換えてディスクをアクセスします。

3.8.8　CONFIG.SYSファイルによるデバイスドライバの登録

本章3.2.3でも触れたように、MS-DOSの起動時に、CONFIG.SYSファイルの内容に従ってシステムのインストールが行われます。そのなかのDEVICEコマンド

DEVICE＝デバイスドライバのファイル名

によって、ユーザー独自のデバイスドライバをシステムに登録することができます（詳しくは、7章参照）。この機能は、MS-DOSの大きな特長の1つでもあります。

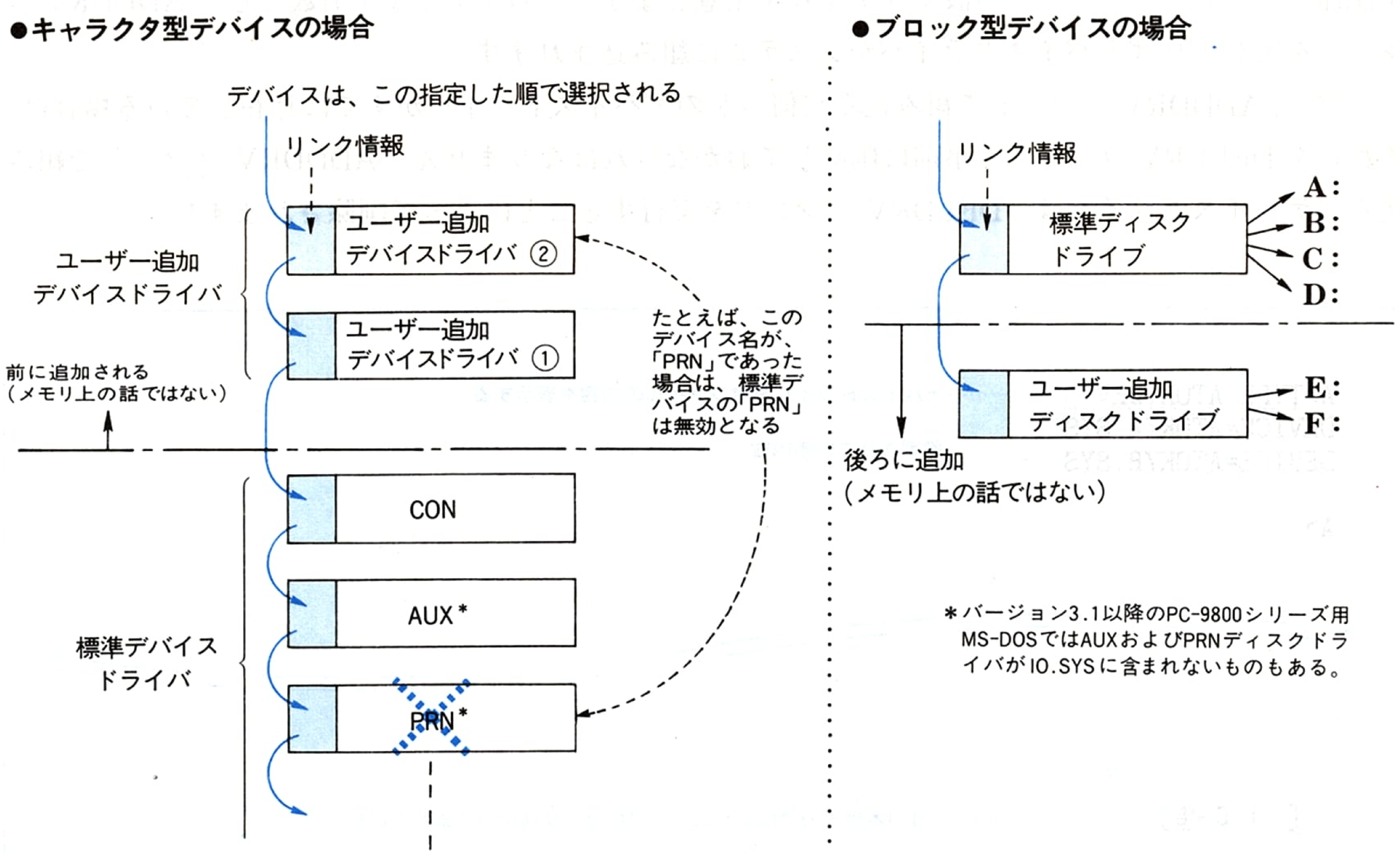
CONFIG.SYSファイルによってシステムに登録される、ユーザー独自のデバイスドライバは、IO.SYS内の標準デバイスドライバとまったく同じ形式で、その立場もまったく対等であり、MSDOS.SYSからコマンドを受け取る方法や、結果を返す方法も同じです。

キャラクタ型デバイスの場合は、もしユーザー独自のデバイスドライバのデバイスファイル名が、標準のデバイスファイル名と同じであれば、その標準ドライバと完全に置き換わります。つまり、標準のドライバは無効となり、代わりにユーザーの登録したドライバが、同じ名前のデバイスドライバとして有効になるのです。たとえば、標準のCONデバイスやPRNデバイスの機能を、ユーザー仕様

のものに変更したければ、ユーザーが作成したドライバのデバイスファイル名を、CON や PRN とすることによって、それと置き換えることができます。

ブロック型デバイスの場合は、ユーザー独自のドライバは、標準のドライブの後ろに追加される形となり、それに続くドライブ番号を持ちます。つまり、標準のドライブ以外のドライブを簡単に「追加」することができるのです。

ここで大切なことは、変更あるいは追加したデバイスドライバは、システムに登録されるため、外部プログラムからは、それらの周辺装置やドライブを、標準のものとまったく同じように、システムを通してアクセスできることです。これらの新しい周辺装置やドライブを追加、変更するのに、MS-DOS 自身にはまったく手を加える必要がなく、デバイスドライバをファイルの形式で用意し、それを DEVICE コマンドで CONFIG.SYS ファイルに記述しておけばよいのです。このような状態を図 3.40 に示しますので、再度確認しておいてください。


●キャラクタ型デバイスの場合
デバイスは、この指定した順で選択される
リンク情報
ユーザー追加
デバイスドライバ
ユーザー追加 デバイスドライバ ②
ユーザー追加 デバイスドライバ ①
前に追加される
（メモリ上の話ではない）
たとえば、このデバイス名が、「PRN」であった場合は、標準デバイスの「PRN」は無効となる
CON
AUX *
PRN *
標準デバイス
ドライバ
●ブロック型デバイスの場合
リンク情報
標準ディスク ドライブ
A:
B:
C:
D:
ユーザー追加 ディスクドライブ
E:
F:
後ろに追加
（メモリ上の話ではない）

＊バージョン3.1以降のPC-9800シリーズ用MS-DOSではAUXおよびPRNディスクドライバがIO.SYSに含まれないものもある。

図 3.40 CONFIG.SYS ファイルによるデバイスドライバの登録

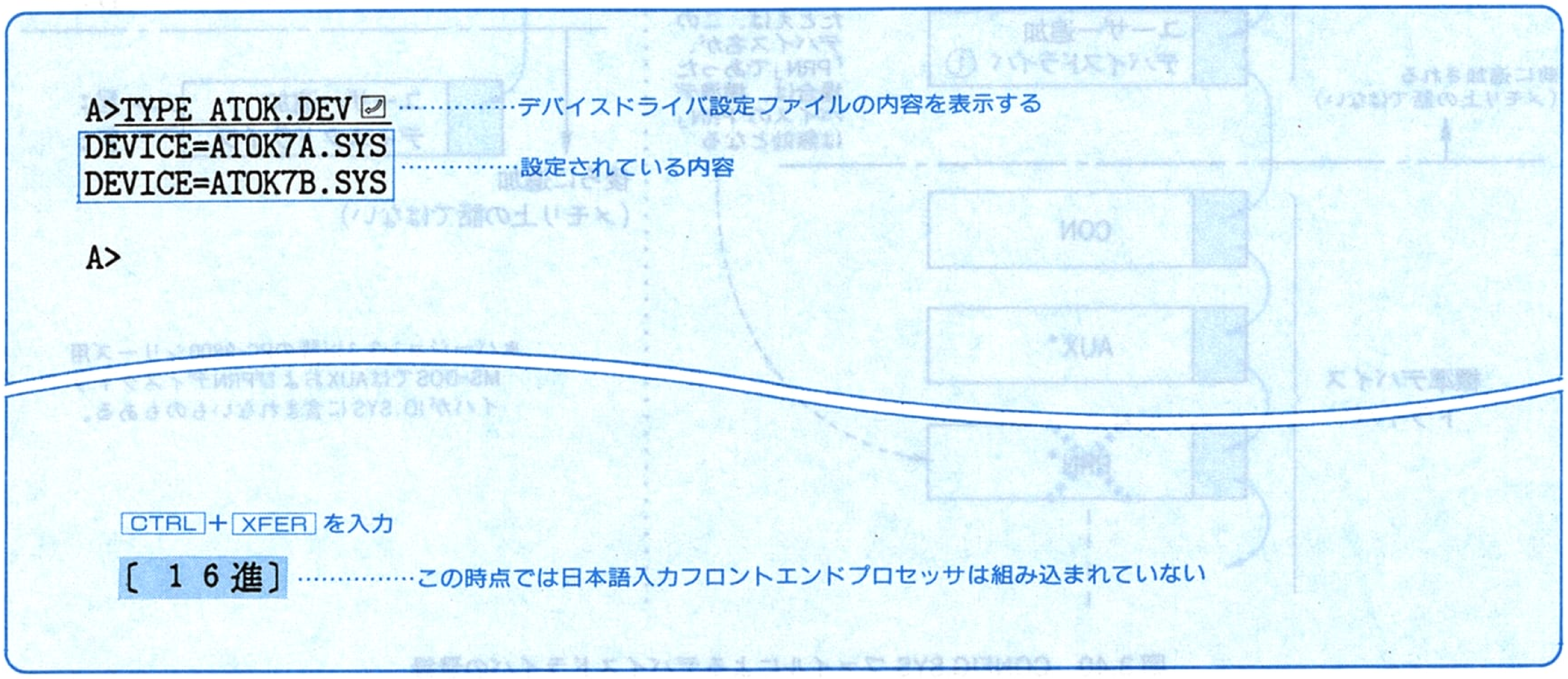
3.8.9　ADDDRV、DELDRVコマンドによるデバイスドライバの登録と削除

CONFIG.SYSファイルにデバイスドライバのファイル名を記述しておくことにより、システム起動時にそのデバイスドライバをシステムに組み込むことができます。逆に言えば、デバイスドライバはシステム起動時にしか組み込むことができません。

ところがMS-DOSバージョン3.*x*からはシステム起動後でもデバイスドライバ(キャラクタ型デバイスドライバに限る)を組み込むことができるようになりました。また、組み込んだデバイスドライバは削除することができるため、システムを再起動することなく別のデバイスドライバに切り換えることが可能です。

デバイスドライバを組み込むコマンドとしてADDDRV、組み込んだデバイスドライバを再び切り離すコマンドとしてDELDRVが用意されています。ADDDRVを使ってデバイスドライバを組み込むためには、**図3.41**に示すようにCONFIG.SYSの場合と同じようなデバイスドライバのファイル名を記述した、デバイスドライバ設定ファイルを用意します。そのファイルを引数としてADDDRVコマンドを実行すればデバイスドライバがシステムに組み込まれます。

ただし、ADDDRVコマンドで組み込んだ何らかのデバイスドライバがすでに存在している場合は、それらをDELDRVコマンドで事前に削除しておかなければなりません。ADDDRVコマンドで組み込んだデバイスドライバは、DELDRVコマンドを実行することによって削除されます*。



```
A>TYPE ATOK.DEV⏎
DEVICE=ATOK7A.SYS
DEVICE=ATOK7B.SYS

A>
```

……デバイスドライバ設定ファイルの内容を表示する

……設定されている内容

CTRL＋XFER を入力

```
[ 1 6進]
```

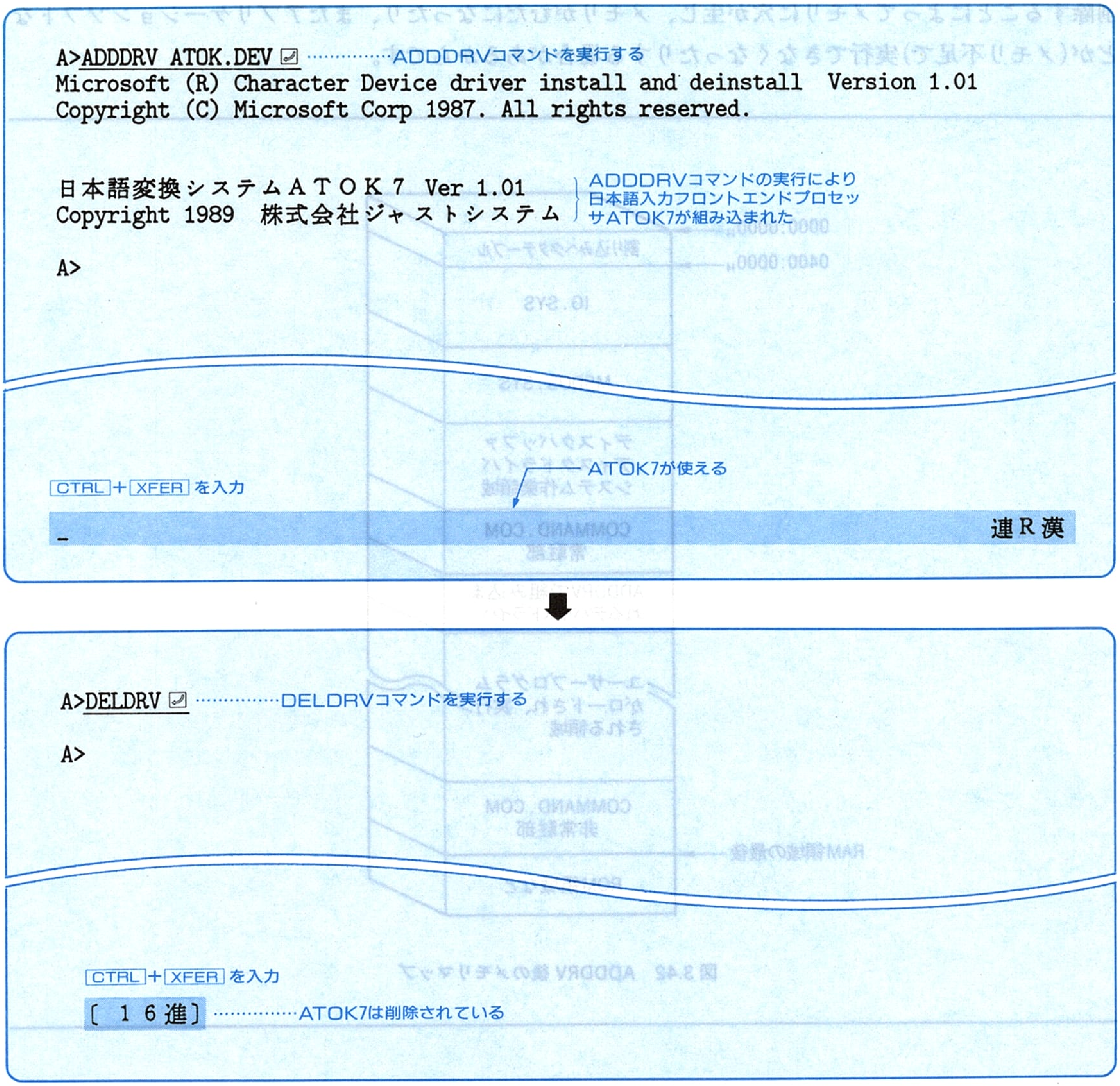
……この時点では日本語入力フロントエンドプロセッサは組み込まれていない

— 図3.41 —

(次ページに続く)

---

* ADDDRV、DELDRVコマンドの詳しい使い方については『実用MS-DOS』を参照のこと。



```
A>ADDDRV ATOK.DEV⏎
Microsoft (R) Character Device driver install and deinstall  Version 1.01
Copyright (C) Microsoft Corp 1987. All rights reserved.


日本語変換システムＡＴＯＫ７ Ver 1.01
Copyright 1989  株式会社ジャストシステム

A>
```

ADDDRVコマンドを実行する

ADDDRVコマンドの実行により日本語入力フロントエンドプロセッサATOK7が組み込まれた

CTRL＋XFER を入力

```
_                                                              連Ｒ漢
```

ATOK7が使える

```
A>DELDRV⏎

A>
```

DELDRVコマンドを実行する

CTRL＋XFER を入力

```
［１６進］
```

ATOK7は削除されている

図 3.41　ADDDRV と DELDRV

ADDDRV コマンドで組み込めるデバイスドライバはキャラクタ型デバイスだけです。つまり、カナ漢字変換フロントエンドプロセッサやマウスドライバを組み込むということはできますが、RAM ディスクドライバやディスクキャッシュ、EMS(拡張メモリ)などを組み込むことはできません。

ADDDRV コマンドでデバイスドライバを組み込んだ状態のメモリマップは **図 3.42** のようになります。この図からわかるように、デバイスドライバ組み込み後に常駐型のユーザープログラムをロードして実行することは避けるべきです。というのは、それ以前に組み込まれているデバイスドライバを

削除することによってメモリに穴が生じ、メモリがむだになったり、またアプリケーションソフトなどが(メモリ不足で)実行できなくなったりする場合があるからです。

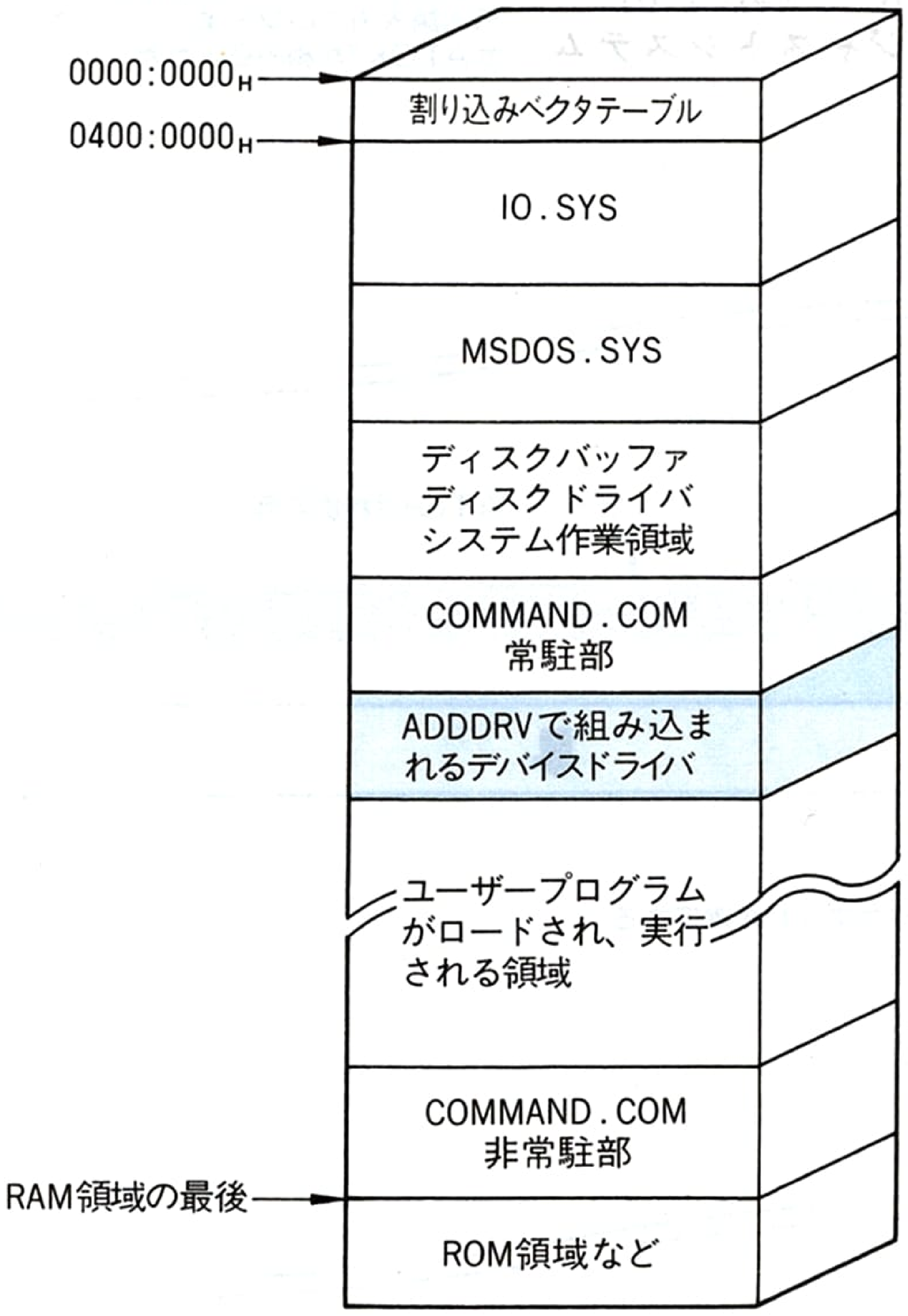

0000:0000H
0400:0000H
割り込みベクタテーブル
IO.SYS
MSDOS.SYS
ディスクバッファ
ディスクドライバ
システム作業領域
COMMAND.COM
常駐部
ADDDRVで組み込まれるデバイスドライバ
ユーザープログラムがロードされ、実行される領域
COMMAND.COM
非常駐部
RAM領域の最後
ROM領域など

図3.42　ADDDRV後のメモリマップ

3.8.10　デバイスドライバを作成するメリット

ユーザー仕様のデバイスドライバを登録することにより、標準のデバイスドライバと置き換えたり、新たなドライブを追加したりすることができますが、そのほかに、デバイスドライバを登録することのメリットはあるのでしょうか。新しい装置を操作するには、なにもデバイスドライバを作って登録しなくても、ユーザープログラムから直接操作してもよいのです。その装置を複数のユーザープログラムで利用するとしても、そのプログラムをライブラリ化しておけば、効率よくプログラミングできます。では、このようにユーザープログラムから直接操作するものと、デバイスドライバとしてシステムを介して操作するものとでは、どこが異なるのでしょう。

周辺装置の多くは、入出力のタイミングをハードウェア的な割り込みで行う方が効率がよく、より確実な動作が期待できます。ただし、割り込み処理を行うには、割り込み処理ルーチンが常にメモリ上に存在していなければなりません。というのは、ユーザープログラムを実行しているときのみ、それらの割り込みが発生するとは限らないからです。COMMAND.COMのプロンプト時でも割り込みは発生することがあり、このために、その周辺装置のドライバプログラムを常駐させるという手法もないわけではありませんが、プログラムや全体の運用がめんどうになり、また環境やPSPの領域、その他のメモリが多少むだになってしまいます。

一方、デバイスドライバとしてシステムに登録した場合はどうでしょう。デバイスドライバは、普通、システムの起動時にシステムに組み込まれ、電源を落とすまで常駐します。また、外部プログラムから周辺装置を直接操作するのに比べて、プログラムの汎用性が高く、異なる機種で同じような周辺装置を使う場合などは、ハードウェアを直接操作する部分をデバイスドライバとして分離することにより、ユーザープログラム自体はまったく同じものが利用できることになります。

たとえば、周辺装置としてグラフィック・ディスプレイを考えてみましょう。グラフィック・ディスプレイをユーザープログラムから直接操作する場合は、そのハードウェアは機種ごとに異なりますので、そのプログラムも機種ごとに異なったものを書かなければなりません(**図3.43**)。

ところが、グラフィック・ディスプレイのハードウェアを直接操作する部分のプログラムを分離し、その部分をデバイスドライバとして機種ごとに用意してシステムに登録すれば、ユーザープログラムは各機種で共通に使えることになります*(**図3.44**参照)。

---

* 機種によっては、グラフィック・デバイスドライバの付属しているシステムもあり(バージョン3.*x*のPC-9800シリーズなど)、MS-DOSからグラフィックが扱いやすくなっているが、いまのところ仕様が統一されていないので、各機種でユーザープログラムを共通にはできない。

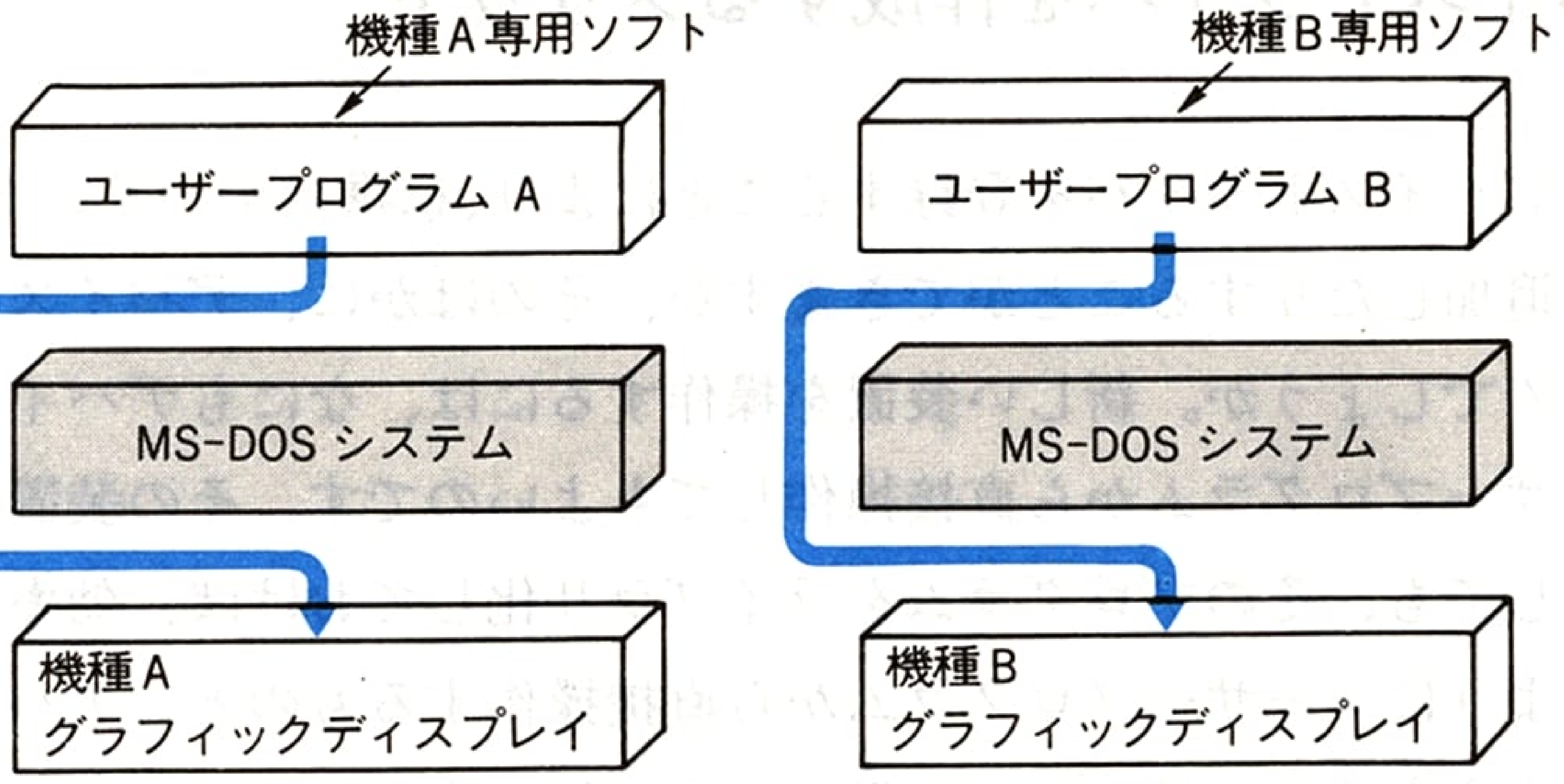

機種A専用ソフト
ユーザープログラム A
MS-DOS システム
機種A
グラフィックディスプレイ
機種B専用ソフト
ユーザープログラム B
MS-DOS システム
機種B
グラフィックディスプレイ

図3.43　グラフィック・ディスプレイをユーザープログラムで直接操作する

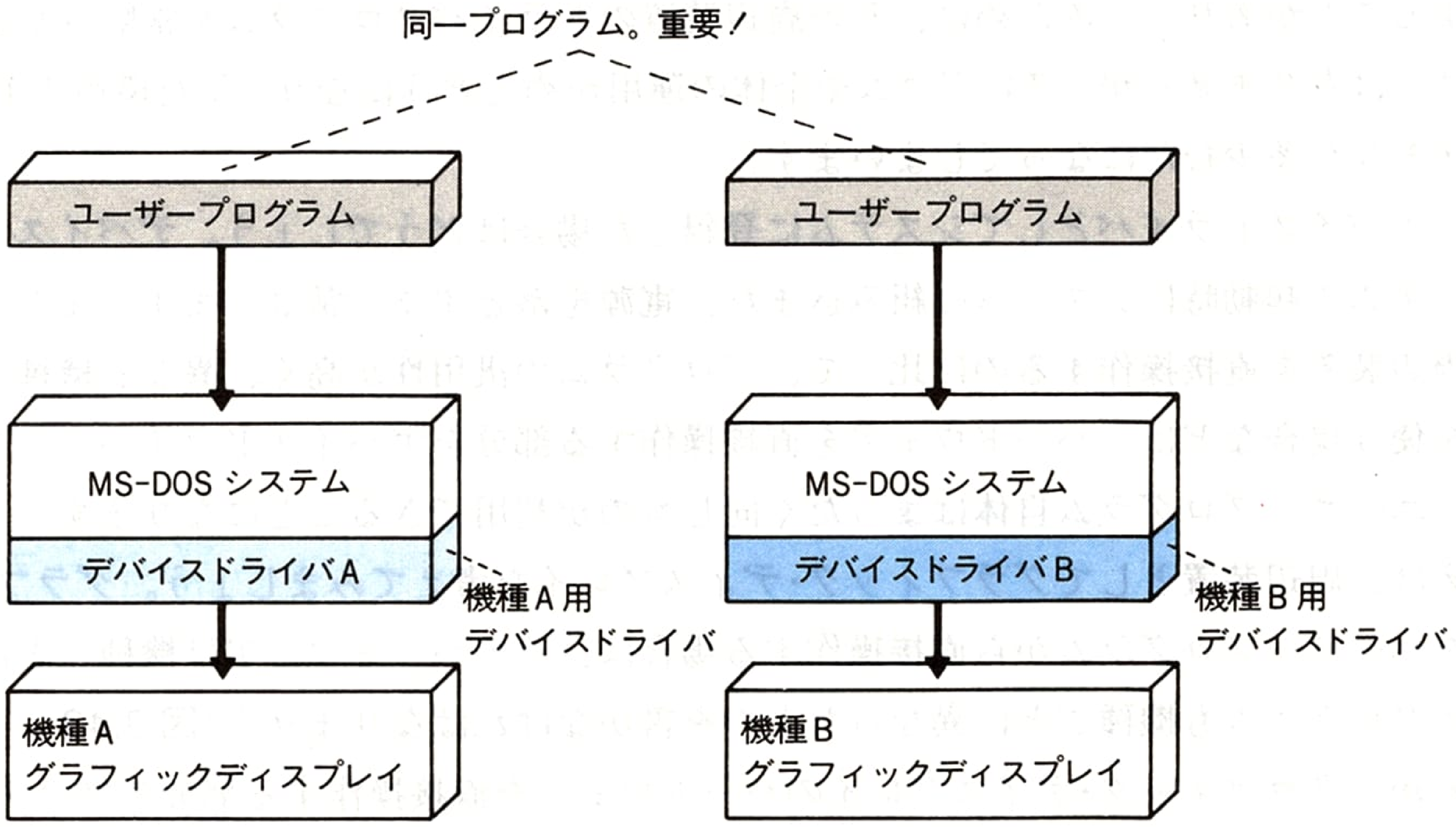
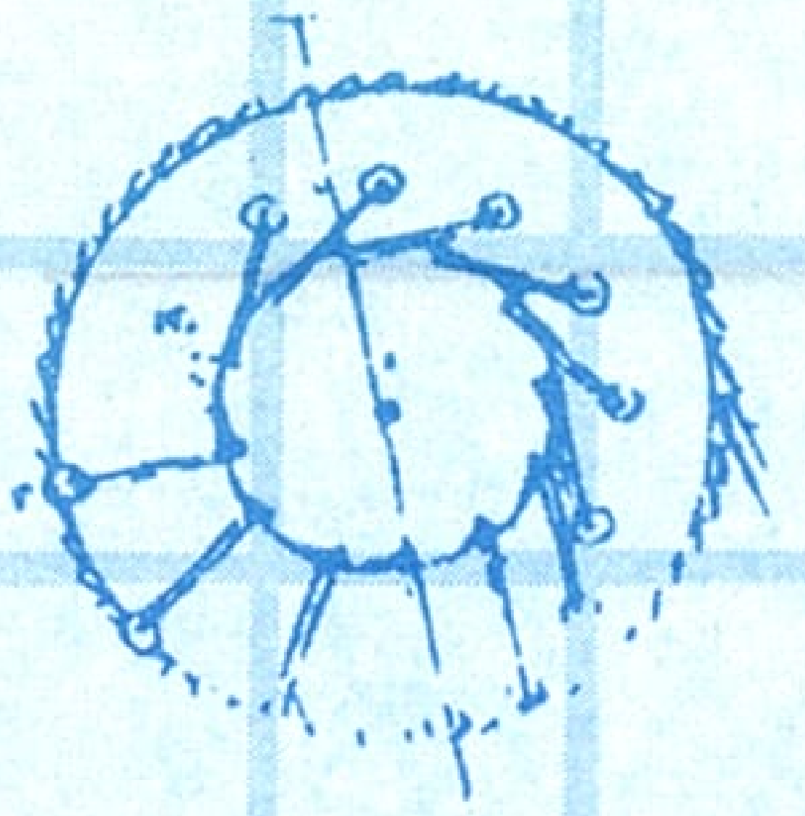

同一プログラム。重要！
ユーザープログラム
MS-DOS システム
デバイスドライバA
機種A用
デバイスドライバ
機種A
グラフィックディスプレイ
ユーザープログラム
MS-DOS システム
デバイスドライバB
機種B用
デバイスドライバ
機種B
グラフィックディスプレイ

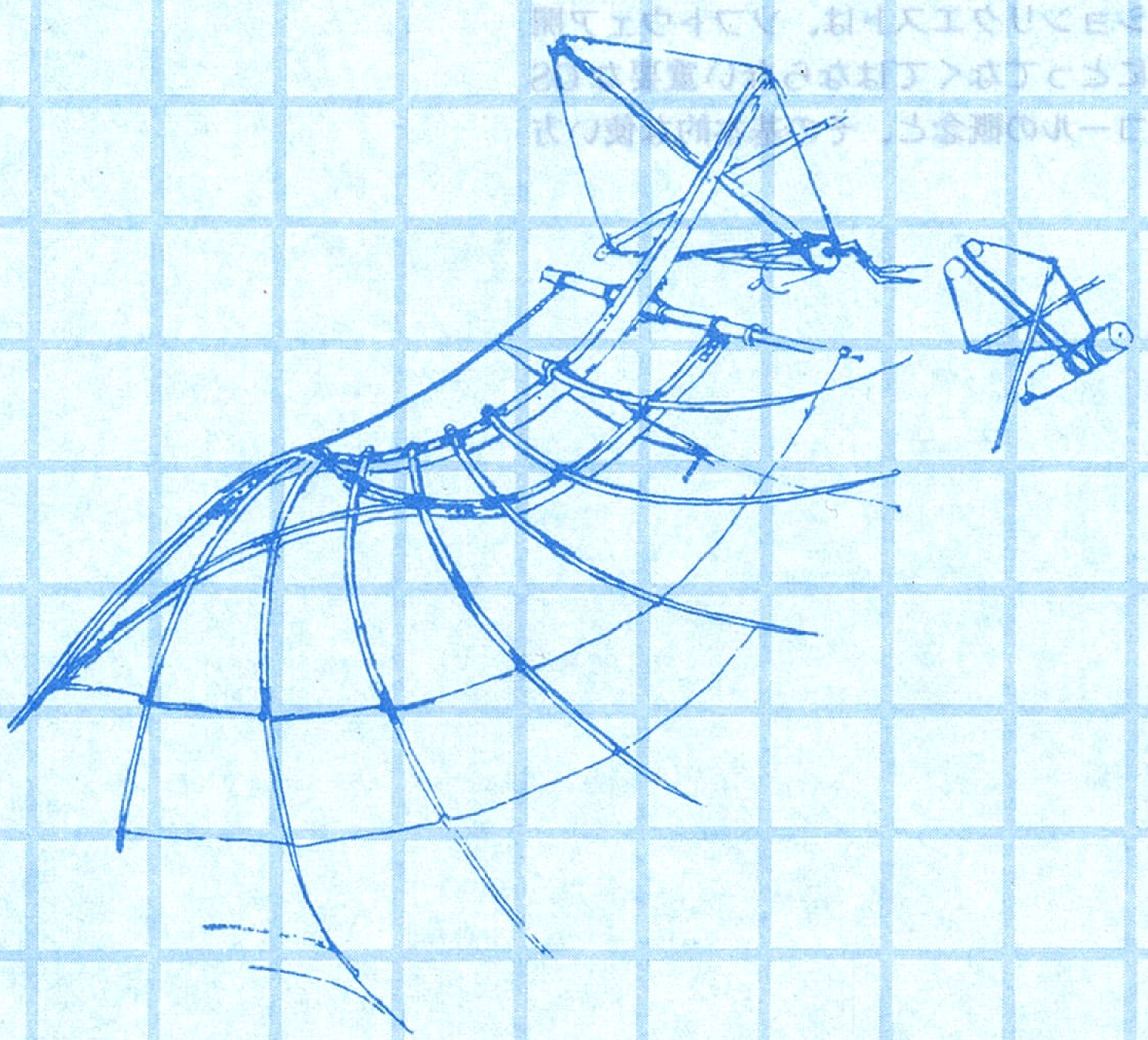
図3.44　グラフィックディスプレイをデバイスドライバを介してユーザープログラムで操作する

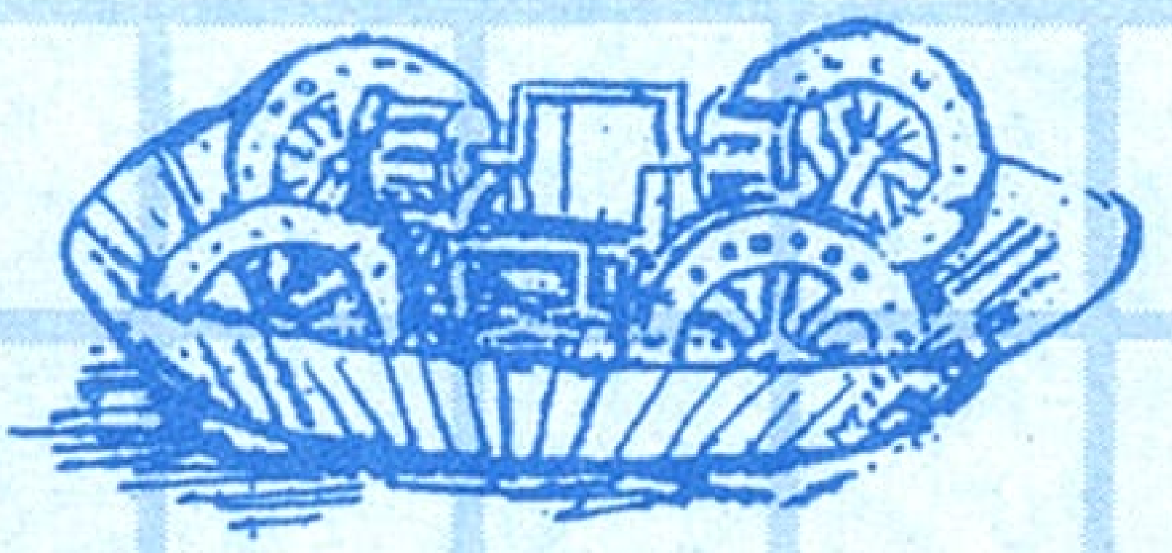
なお7章では、グラフィックディスプレイを操作する簡単なデバイスドライバを作成し、これをユーザープログラムからアクセスしてフリーハンドの絵を書くプログラムを作成します。また、デバイスドライバを使用せず、ユーザープログラムで直接グラフィックディスプレイを操作する、同じ機能を持つプログラムも作成します。本章で述べたことの実際面のさらに詳しい解説は、7章を参照してください。

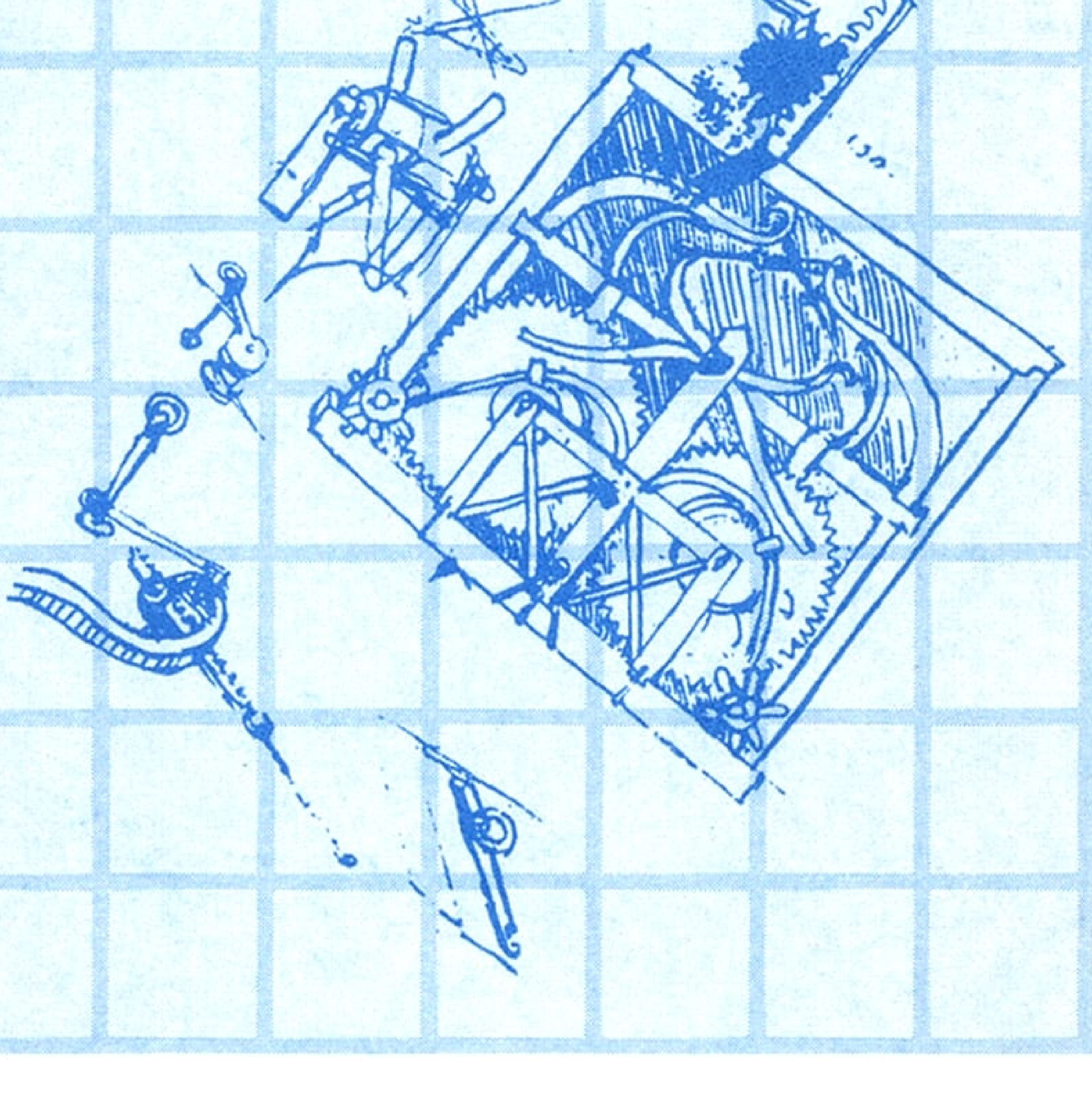


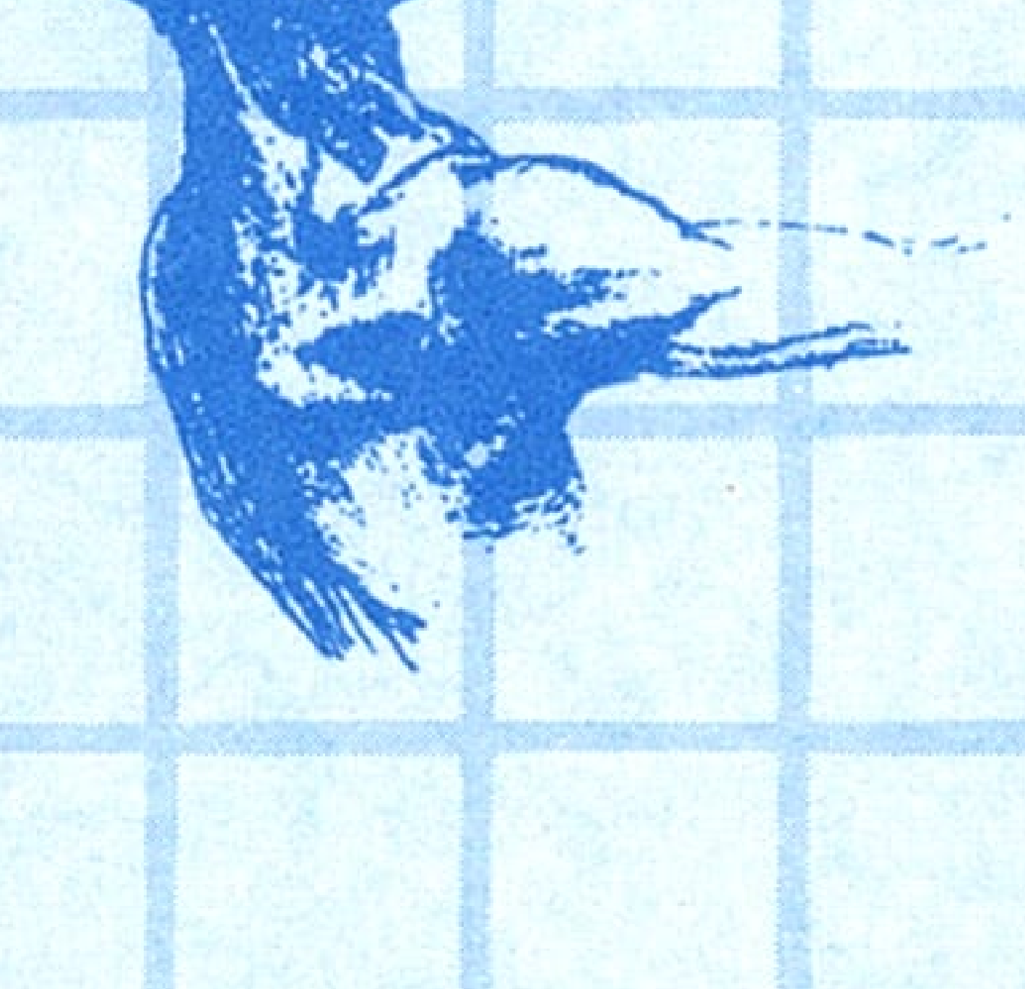
4章

システムコールとソフトウェア割り込み









MS-DOSは、16ビットの事実上の標準OSとして広く普及し、MS-DOS上て走る多種多様なソフトウェアが大量に市販されるようになりました。それというのも、多くのソフトウェアハウスによってMS-DOS上のソフトウェアの開発競争が行われ、品質の高い製品が次々と世に送り出されているおかげです。このようにソフトウェアハウス各社が、高度なソフトウェア製品を開発できる背景には、第一にMS-DOSを取り巻く環境が充実してきたことが挙げられますが、それらのすべては本章のテーマである「システムコール」と「ファンクションリクエスト」の上に成り立っています。

システムコール、とくにファンクションリクエストは、ソフトウェア開発をはじめとするMS-DOSの応用にとってなくてはならない重要なOSの機能です。本章は、このシステムコールの概念と、その基本的な使い方について解説しましょう。

4.1 システムコールとファンクションリクエストの概念

システムコールは、MS-DOSが内蔵している多くのソフトウェア機能を開発者に利用しやすい形式で提供します。つまり、OS内部の各種の機能を**システムコール**という形で公開し、開発者はユーザープログラムにおいて、その中の必要な機能を呼び出して利用することができるのです。MS-DOS「システム」内の各種の機能をユーザープログラムで「コール」する、これがシステムコールです。

システムコールのほとんどの機能は、後述の「INT　21H」による**ファンクションリクエスト**と呼ばれるシステムのコール法として実現されています。ファンクションリクエストは、その名のとおり、OS内の「機能(ファンクション)の要求」をするための、ソフトウェア開発にはなくてはならないOSの重要な機能です。

それぞれの機能を呼び出す手順や、結果の出力形式は定型化されており、コンピュータの機種や個々のハードウェアに依存する部分は表面には現れていません。つまり、ファンクションリクエストはすべてのMS-DOSマシンに共通して提供されている機能であり、ファンクションリクエストを利用したユーザープログラムはコンピュータの機種に依存せずに全MS-DOSマシンでの互換性が保たれるのです。

このファンクションリクエストは、MS-DOSバージョン2.xで約70種類、3.xでは約100種類と、非常に豊富に用意されています。システムコールで提供されている全ファンクションを、機能別に分類して**表4.1**に示しましょう。

ファイルの管理

ファンクションナンバー	機　能
3C$_{H}$	ハンドルの作成
3D$_{H}$	ハンドルのオープン*
3E$_{H}$	ハンドルのクローズ
3F$_{H}$	リードハンドル
40$_{H}$	ライトハンドル
42$_{H}$	ファイルポインタの移動
45$_{H}$	ファイルハンドルの二重化
46$_{H}$	ファイルハンドルの強制二重化
5A$_{H}$	テンポラリファイルの作成
5B$_{H}$	新規ファイルの作成

ディレクトリの管理

ファンクションナンバー	機　能
39$_{H}$	ディレクトリの作成
3A$_{H}$	ディレクトリの削除
3B$_{H}$	カレントディレクトリの変更
41$_{H}$	ディレクトリエントリの削除
43$_{H}$	ファイルの属性を得る／セットする
47$_{H}$	カレントディレクトリを得る
4E$_{H}$	最初に一致するファイル名の検索
4F$_{H}$	次に一致するファイル名の検索
56$_{H}$	ディレクトリエントリの変更
57$_{H}$	ファイルの作成日時を得る／セットする

青い文字のファンクションは、バージョン3.1で新しく追加されたもの
*印のファンクションは、バージョン3.1で機能が拡張/変更されたもの
**印のファンクションは、バージョン3.1で機能が公開されたもの
***印のファンクションは、バージョン3.3で新しく追加されたもの

— 表4.1 — (次ページに続く)

キャラクタデバイスの入出力

ファンクションナンバー	機能
01H	キーボードからの1文字入力とエコー表示
02H	ディスプレイへの1文字出力
03H	補助入力
04H	補助出力
05H	プリンタへの1文字出力
06H	直接コンソール入出力
07H	直接コンソール入力
08H	キーボード入力(エコーなし)
09H	文字列のディスプレイ出力
0AH	バッファードキーボード入力
0BH	キーボードステータスのチェック
0CH	バッファを空にして、キーボード入力

デバイス管理

ファンクションナンバー	機能
4400H[注1]	IOCTL[注2]データを得る
4401H	IOCTLデータをセットする
4402H	IOCTLキャラクタを送る
4403H	IOCTLキャラクタを受け取る
4404H	IOCTLブロックを送る
4405H	IOCTLブロックを受け取る
4406H	入力ステータスのチェック
4407H	出力ステータスのチェック
4408H	IOCTLの交換可能性のチェック
440CH	ジェネリックIOCTL(ハンドル用)***
440DH	ジェネリックIOCTL(ブロック型デバイス用)***
440EH	論理ドライブマップを得る***
440FH	論理ドライブマップを設定する***

メモリの管理

ファンクションナンバー	機能
48H	メモリの割り当て
49H	割り当てられたメモリの解放
4AH	割り当てられたメモリブロックの変更
58H	メモリの割り当て方法を得る／セットする**

プロセスの管理

ファンクションナンバー	機能
26H	PSPの作成**
31H	キーププロセス
4B00H	プログラムのロードと実行
4B03H	オーバーレイのロード
4CH	プロセスの終了
4DH	子プロセスからリターンコードを得る
62H	PSPを得る

各種のシステム管理

ファンクションナンバー	機能
0DH	ディスクのリセット
0EH	ディスクの選択*
19H	カレントドライブを得る
1AH	ディスク転送アドレスのセット
1BH	カレントドライブのデータを得る**
1CH	ドライブのデータを得る**
25H	割り込みベクタのセット
29H	ファイル名の解析
2AH	日付を得る[注3]
2BH	日付のセット[注3]
2CH	時刻を得る
2DH	時刻のセット
2EH	ベリファイフラグのセット／リセット
2FH	ディスク転送アドレスを得る
30H	MS-DOSのバージョンナンバーを得る
33H	Ctrl-Cのチェック
35H	割り込みベクタを得る
36H	ディスクの空き領域を得る
38H	国別情報を得る／セットする*
54H	ベリファイの状態を得る
59H	拡張エラーコードを得る

ファイル・シェアリング関係

ファンクションナンバー	機能
3DH	ハンドルのオープン*
440BH	IOCTLリトライ回数のセット
5C00H	ファイルアクセスのロック
5C01H	ファイルアクセスのロック解除

MS-Networks関係

ファンクションナンバー	機能
4409H	IOCTLリディレクトブロックのチェック
440AH	IOCTLリディレクトハンドルのチェック
5E00H	マシン名を得る
5E02H	プリンタセットアップ
5F02H	割り当てリストのエントリを得る
5F03H	割り当てリストのエントリを作成する
5F04H	割り当てリストのエントリを取り消す

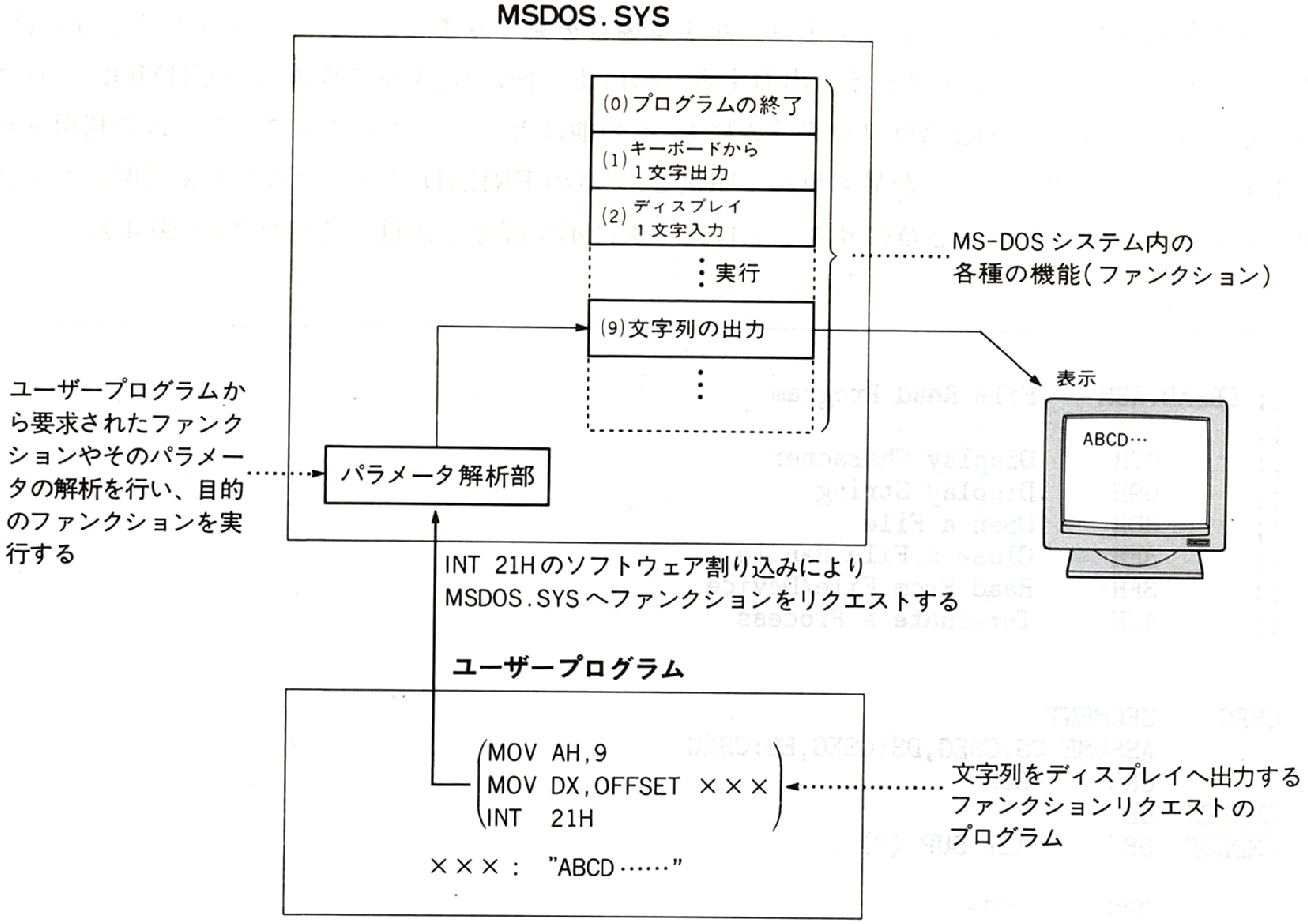
注1　ファンクションナンバーが4桁のもの、たとえば「4400H」であれば、AHレジスタ＝44H、ALレジスタ＝00Hとなる。

注2　IOCTL(I/Oコントロール)。デバイスをコントロールするために、デバイス自身と入出力するコントロールデータ。

注3　バージョン3.1では、年は1980～2079
バージョン3.3では、年は1980～2099

表4.1　ファンクションリクエストで提供されているファンクション一覧

図 4.1 はファンクションリクエストの概念を表したもので、ユーザープログラムの一部でファンクションリクエストを利用している状態を示しています。


MSDOS.SYS
(0)プログラムの終了
(1)キーボードから1文字出力
(2)ディスプレイ1文字入力
実行
(9)文字列の出力
MS-DOS システム内の各種の機能(ファンクション)
表示
ABCD…
ユーザープログラムから要求されたファンクションやそのパラメータの解析を行い、目的のファンクションを実行する
パラメータ解析部
INT 21H のソフトウェア割り込みにより MSDOS.SYS へファンクションをリクエストする
ユーザープログラム
文字列をディスプレイへ出力するファンクションリクエストのプログラム

```
MOV  AH,9
MOV  DX,OFFSET ×××
INT  21H
××× :  "ABCD……"
```

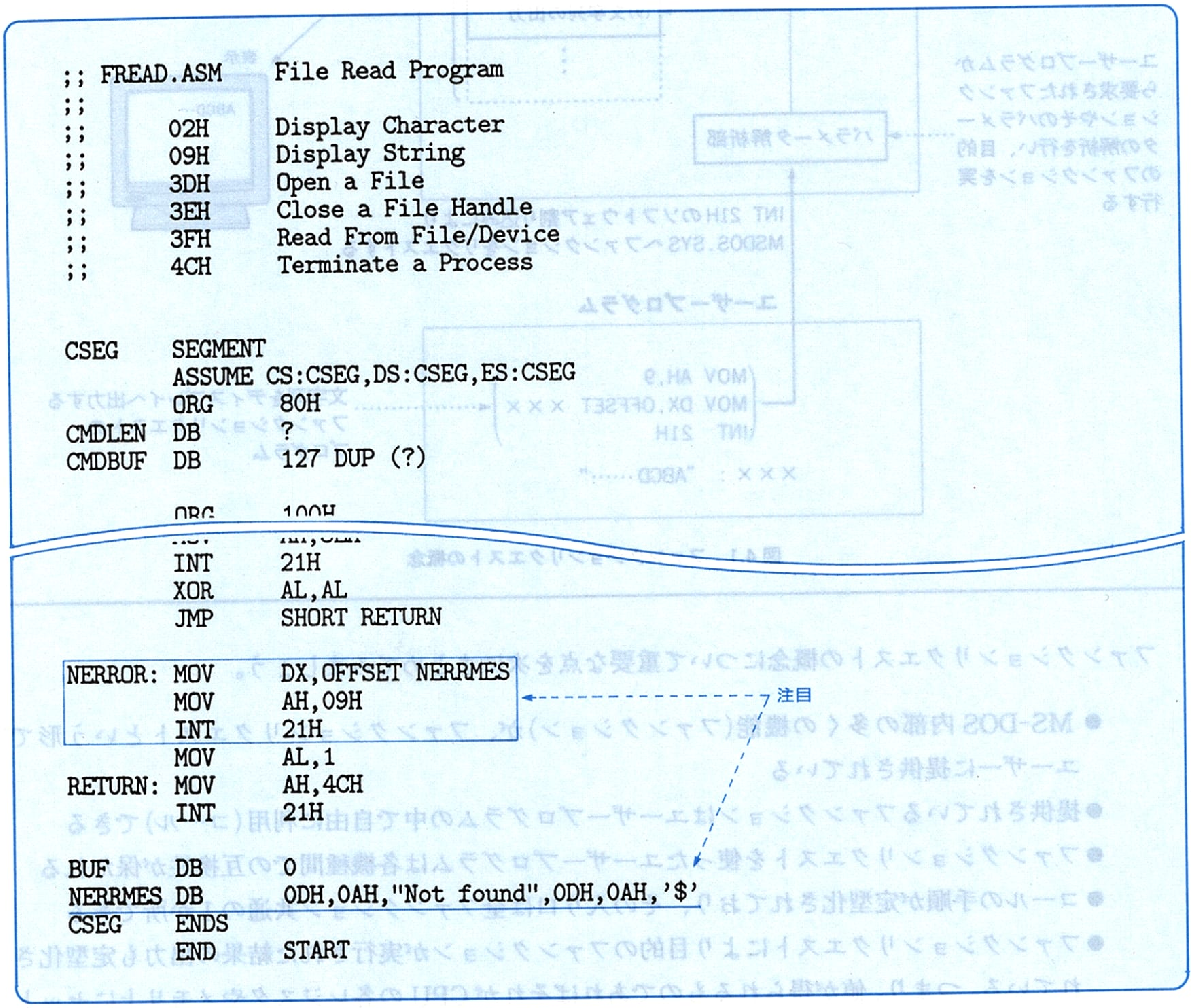
図 4.1　ファンクションリクエストの概念

ファンクションリクエストの概念について重要な点を次にまとめてみましょう。

- MS-DOS 内部の多くの機能(ファンクション)が、ファンクションリクエストという形でユーザーに提供されている
- 提供されているファンクションはユーザープログラムの中で自由に利用(コール)できる
- ファンクションリクエストを使ったユーザープログラムは各機種間での互換性が保たれる
- コールの手順が定型化されており、その入り口は全ファンクション共通の1か所である
- ファンクションリクエストにより目的のファンクションが実行された結果の出力も定型化されている。つまり、値が得られるものであればそれが CPU の各レジスタやメモリ上にセットされ、その他のものではそれぞれの動作が行われる

4.2 ファンクションリクエストの使い方の基礎知識

次に、ファンクションリクエストの実例として、最も単純で基本的な機能の1つである「ディスプレイへの文字列出力」のファンクションをコールする場合を見てみましょう。このファンクションは、各種のメッセージなどをディスプレイに出力するためによく使われ、1章で作成したCHMODプログラムにも、2章で作成したFREADプログラムにも、その他ほとんどのサンプルプログラムで利用されています。ここでは、プログラムが最も単純で明快な、2章のFREADプログラムでの使用例をもとに解説します。ソースファイル(2章のリスト2.1)内の次に示す部分に注目してください(図4.2)。

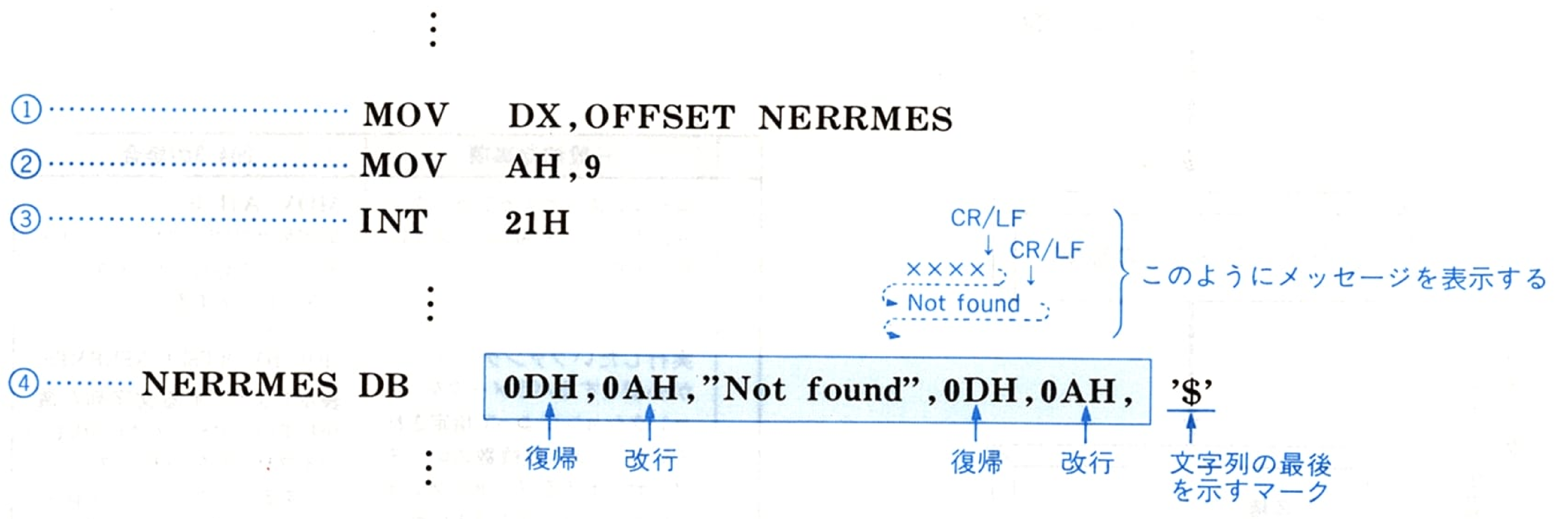


```
;; FREAD.ASM    File Read Program
;;
;;      02H     Display Character
;;      09H     Display String
;;      3DH     Open a File
;;      3EH     Close a File Handle
;;      3FH     Read From File/Device
;;      4CH     Terminate a Process


CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG
        ORG     80H
CMDLEN  DB      ?
CMDBUF  DB      127 DUP (?)

        ORG     100H
```

```
        INT     21H
        XOR     AL,AL
        JMP     SHORT RETURN

NERROR: MOV     DX,OFFSET NERRMES
        MOV     AH,09H
        INT     21H
        MOV     AL,1
RETURN: MOV     AH,4CH
        INT     21H

BUF     DB      0
NERRMES DB      0DH,0AH,"Not found",0DH,0AH,’$’
CSEG    ENDS
        END     START
```

図4.2　FREADプログラムにおける文字列出力のファンクションリクエスト

実際のソースファイルから文字列出力のファンクションをコールする部分を取り出し、整理してみましょう。**図4.3**のようなプログラムで文字列出力の機能がコールされています。

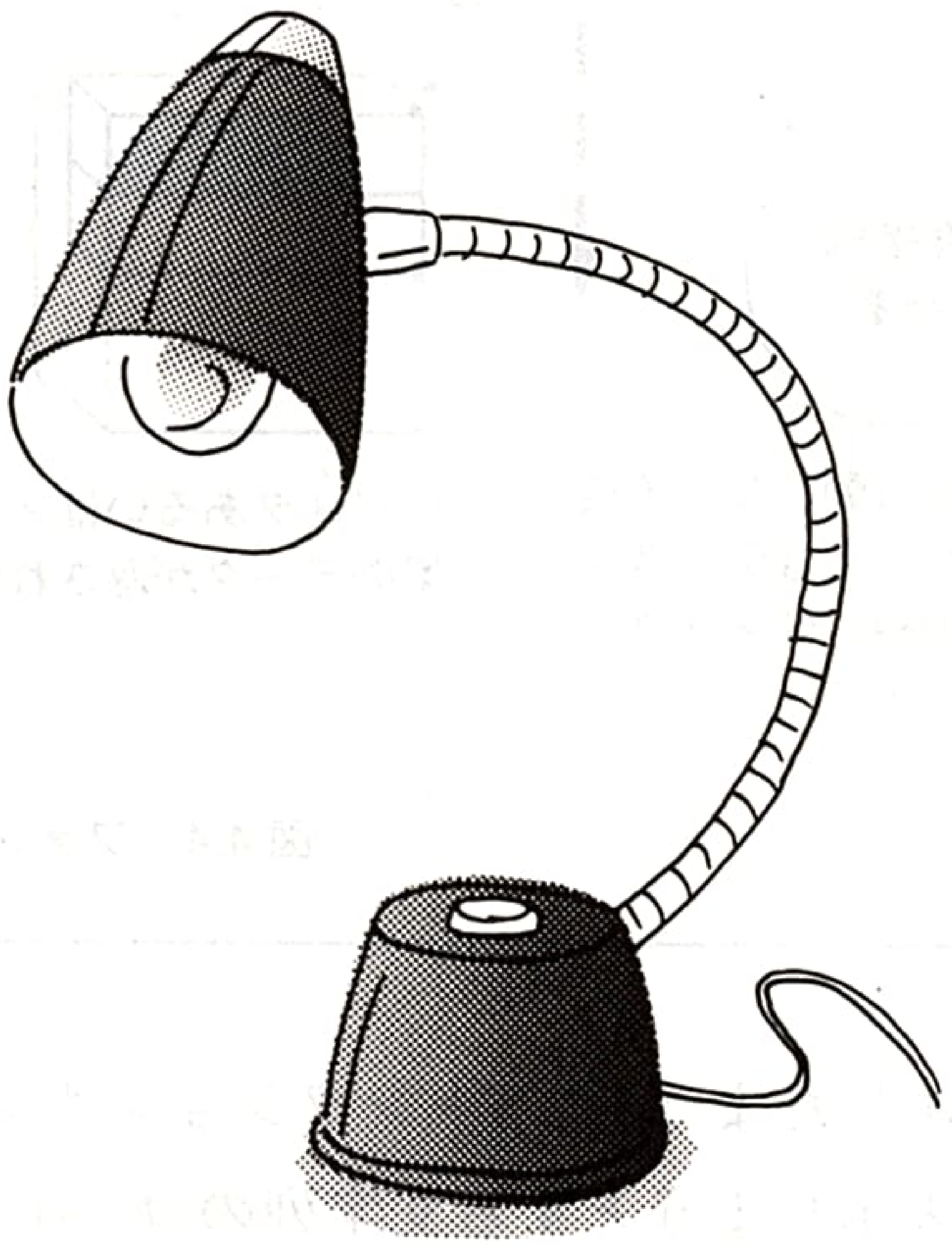


```
                    :
MOV     DX,OFFSET NERRMES
MOV     AH,9
INT     21H
                    :
NERRMES DB   0DH,0AH,"Not found",0DH,0AH, '$'
                    :
```

①
②
③
④
CR/LF
CR/LF
××××
Not found
このようにメッセージを表示する
復帰
改行
復帰
改行
文字列の最後を示すマーク

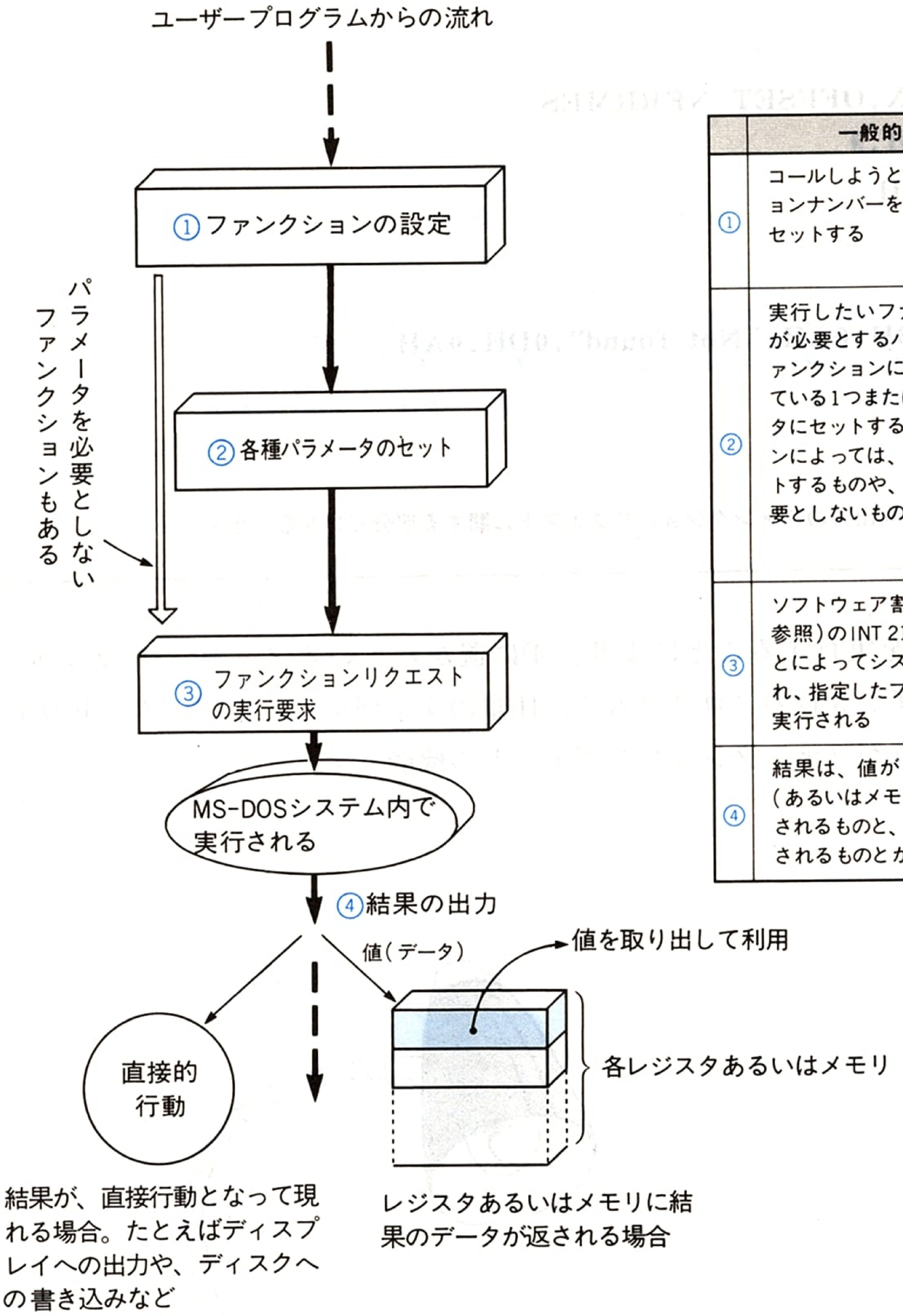
図4.3　図4.2から文字列出力のファンクションリクエストに関する部分だけを取り出す

この①②③の3行のプログラムを実行することにより、④に置かれているメッセージ(文字列)がディスプレイに出力されます。わずか3行のプログラムで、任意の文字列をディスプレイに出力するルーチンが実現できるのです。これがファンクションリクエストの機能です。



■ 目的のファンクションをコールする手順

ファンクションリクエストで目的のファンクションをコールする手順を図 4.4 に示します。


ユーザープログラムからの流れ
①ファンクションの設定
パラメータを必要としないファンクションもある
②各種パラメータのセット
③ファンクションリクエストの実行要求
MS-DOSシステム内で実行される
④結果の出力
値（データ）
値を取り出して利用
直接的行動
各レジスタあるいはメモリ
結果が、直接行動となって現れる場合。たとえばディスプレイへの出力や、ディスクへの書き込みなど
レジスタあるいはメモリに結果のデータが返される場合

	一般的な事項	図4.3の場合
①	コールしようとするファンクションナンバーを AH レジスタにセットする	**MOV AH,9** 文字列出力のファンクションナンバーである「9」を AHレジスタにセットする
②	実行したいファンクションが必要とするパラメータを、ファンクションによって指定されている1つまたは複数のレジスタにセットする（ファンクションによっては、メモリ上にセットするものや、パラメータを必要としないものもある）	**MOV DX,OFFSET NERRMES** 表示しようとする文字列が置かれているアドレス(NERRMES)のオフセットをDXレジスタにセットする。セグメントアドレスをセットする DS レジスタは、COM形式のプログラムを作成する場合は操作する必要はなく、そのままの状態でよい
③	ソフトウェア割り込み（3.7.2 参照）の INT 21H を実行することによってシステムがコールされ、指定したファンクションが実行される	**INT 21H** ソフトウェア割り込みINT 21Hによりシステムをコールし、目的のファンクションを実行する
④	結果は、値が CPU のレジスタ（あるいはメモリ上）にセットされるものと、動作として処理されるものとがある	結果はディスプレイに直接表示される

図 4.4　ファンクションリクエストの実行手順

図 4.4 に示したように、ファンクションナンバー9 の文字列出力のファンクションリクエストを実行するには、表示しようとする文字列のコードをメモリ上に置き（文字列の最終には、終わりを示すため

の「$」マーク(コード24H)を置く)、この文字列の先頭のメモリアドレスをDXレジスタにセットしてINT 21Hを実行します。それによってファンクションナンバー9の実行が開始され、メモリ上の文字列を先頭から順に1文字ずつ取り出してはディスプレイに出力し、この動作を最後の「$」が現れるまで続けます。

■ ファンクションリクエストの結果について

ファンクションリクエストは、ファンクションナンバーを選択して必要なパラメータをセットしたあと、ソフトウェア割り込み(内部割り込みとも呼ぶ)のINT 21Hによってシステムがコールされ、目的のファンクションが実行されて結果が得られます。図4.4にも示してありますが、得られた結果は、値を返すファンクションであれば、その値がCPUのレジスタあるいはメモリ上にセットされます。また、値を返すファンクションでないものは、直接そのファンクションによる処理が行われます。さきほど例を示した文字列出力のファンクションリクエストの場合は、値を返す形態ではなく、結果はコンソール(ディスプレイ)に直接表示されます。

次に、ファンクションリクエストを実行した結果、値がCPUのレジスタにセットされて返される形態の実例を示しましょう。最も単純で基本的なファンクションの1つである「キーボードからの1文字入力」の例です。このファンクションナンバーは1ですので、コールは**図4.5**のようになります。

```
 :
MOV  AH,1
INT  21H
 :
```

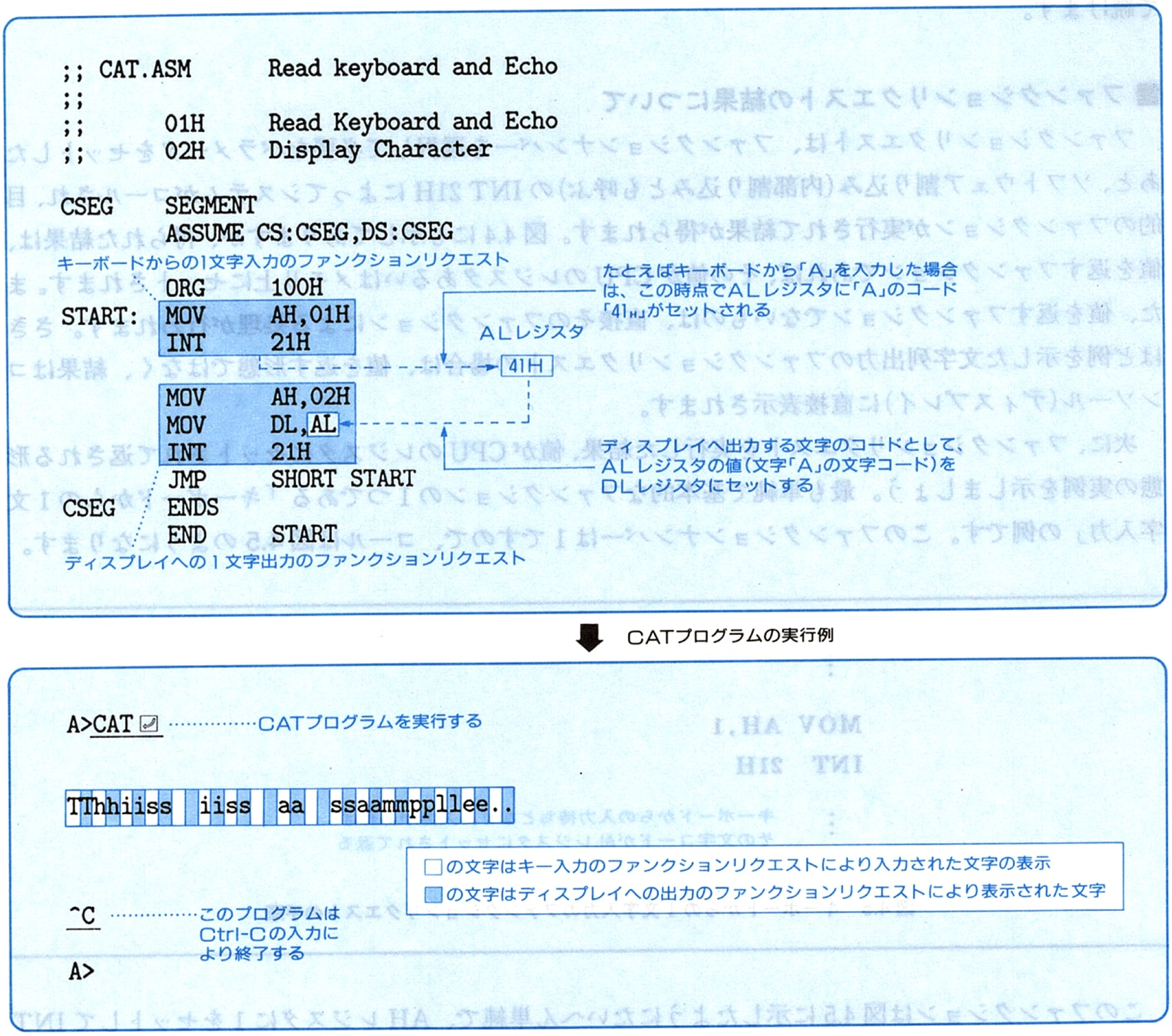
キーボードからの入力待ちとなり、入力があれば
その文字コードがALレジスタにセットされて返る

図4.5　キーボードからの1文字入力のファンクションリクエストの手順

このファンクションは図4.5に示したようにたいへん単純で、AHレジスタに1をセットしてINT 21Hを実行するだけです。パラメータを与える必要はありません。このファンクションがコールされると、キーボードからの入力待ちとなり、何らかの入力があると入力された文字コードがALレジスタにセットされてコールから戻ります。ですから、このALレジスタの値を取り出して、そのあとに続くユーザープログラムで自由に利用すればよいのです。

図4.6に示すのはその簡単な例であり、キーボードからの1文字入力の結果がセットされているALレジスタの値を、そのままディスプレイへの1文字出力のファンクションリクエストに渡して表示させるサンプルプログラムです。なお、このプログラムはアセンブルなどの処理を行えば実行する

ことができます。これをソースファイル「CAT.ASM」として、実行可能なオブジェクトプログラムを作成したあとの実行例も併せて示します。



```
;; CAT.ASM       Read keyboard and Echo
;;
;;      01H      Read Keyboard and Echo
;;      02H      Display Character

CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG
        ORG     100H
START:  MOV     AH,01H
        INT     21H

        MOV     AH,02H
        MOV     DL,AL
        INT     21H
        JMP     SHORT START
CSEG    ENDS
        END     START
```

キーボードからの1文字入力のファンクションリクエスト
たとえばキーボードから「A」を入力した場合は、この時点でALレジスタに「A」のコード「41H」がセットされる
ALレジスタ
41H
ディスプレイへ出力する文字のコードとして、ALレジスタの値（文字「A」の文字コード）をDLレジスタにセットする
ディスプレイへの１文字出力のファンクションリクエスト
CATプログラムの実行例

```
A>CAT⏎

TThhiiss  iiss  aa  ssaammppllee..


^C

A>
```

CATプログラムを実行する
□の文字はキー入力のファンクションリクエストにより入力された文字の表示
▨の文字はディスプレイへの出力のファンクションリクエストにより表示された文字
このプログラムはCtrl-Cの入力により終了する

図4.6　１文字入力のファンクションリクエストの実行結果を、１文字出力のファンクションリクエストに渡すプログラムCAT.ASMと実行例

キーボードからの１文字入力のファンクションリクエストは、結果として返る値が入力された１文字の文字コード（1バイト）なので、ALレジスタのみに結果がセットされますが、その他のファンクションで複数の値を返す場合は、複数のレジスタにそれぞれの値がセットされます。

■ ファンクションリクエスト実行時のエラー

ファンクションリクエストを実行した場合、そのファンクションが正常に実行されたかどうか、もし正常に実行されなかった場合はどのようなエラーがあったのかなどを知ることができます。その概要を**図 4.7** に示しましょう。

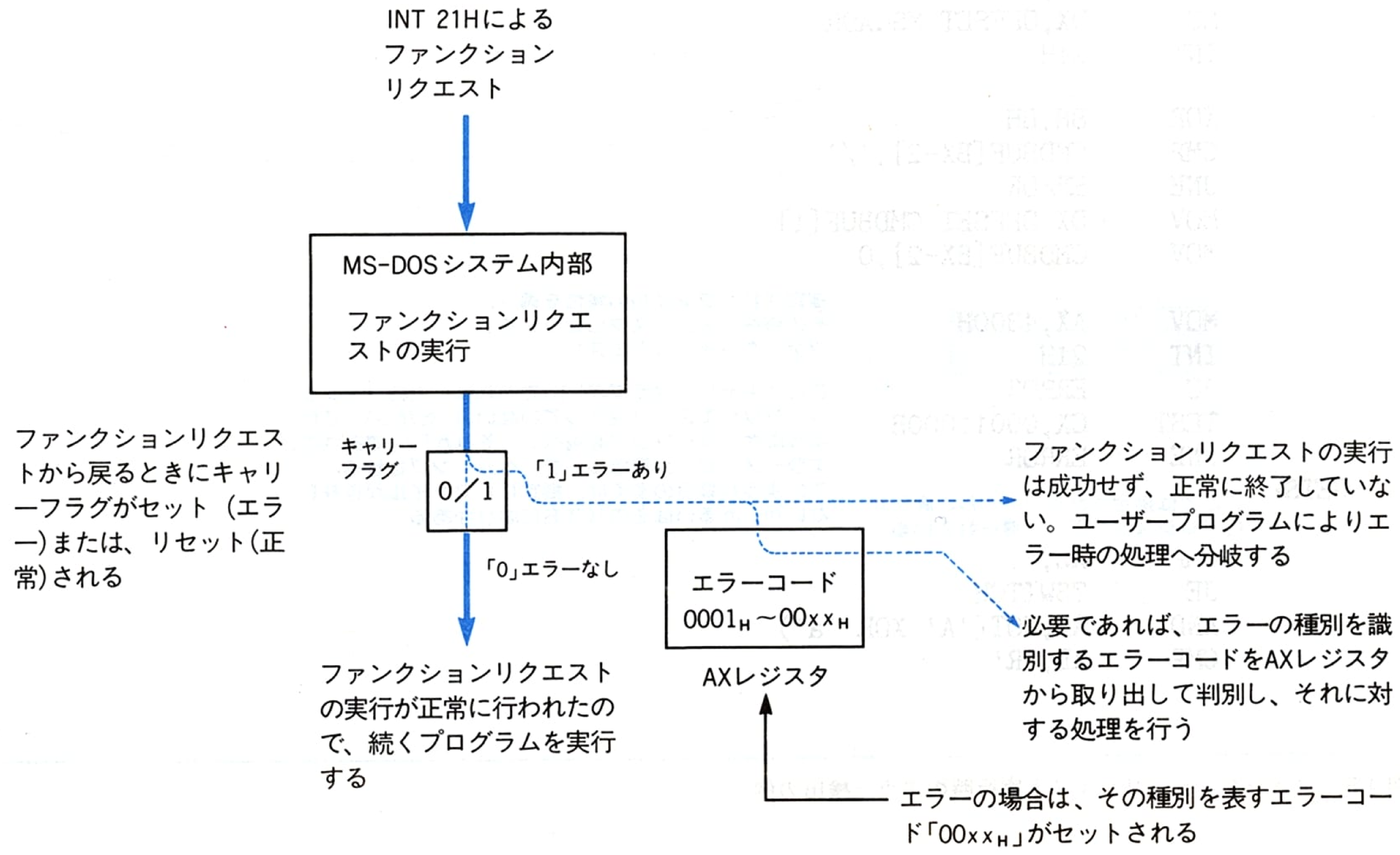

INT 21Hによる
ファンクション
リクエスト
MS-DOSシステム内部
ファンクションリクエストの実行
ファンクションリクエストから戻るときにキャリーフラグがセット（エラー）または、リセット(正常)される
キャリーフラグ
0／1
「1」エラーあり
「0」エラーなし
ファンクションリクエストの実行は成功せず、正常に終了していない。ユーザープログラムによりエラー時の処理へ分岐する
エラーコード
0001H～00xxH
AXレジスタ
必要であれば、エラーの種別を識別するエラーコードをAXレジスタから取り出して判別し、それに対する処理を行う
ファンクションリクエストの実行が正常に行われたので、続くプログラムを実行する
エラーの場合は、その種別を表すエラーコード「00xxH」がセットされる

図 4.7　ファンクションリクエスト実行時のエラーの検出法

コールしたファンクションが正常に実行されたかどうかを知るには、ファンクションリクエストから戻った直後にキャリーフラグを調べます。キャリーフラグが「0」であるなら正常、「1」が立っている場合は何らかのエラーがあったことを示しており、エラーとなった場合は、必要であればユーザープログラムでAXレジスタの値を調べてエラーの種類を確認し、それに対応する処置をすることになります*。

次に、1章のCHMODプログラムで行われているエラーチェックの具体例を見てみましょう（**図 4.8**）。

＊ バージョン1.25以前のファンクションでは、キャリーフラブではなく、ALレジスタの内容によってエラーを示すものがあるが、これらのファンクションは利用すべきではない。

1章で作成したCHMODプログラム

```
;; CHMOD.ASM      CHANGE FILE ATTRIBUTES
;;
;;      09H       Display String
;;      43H       Change Attributes
;;      4CH       Terminate a Process

PRINT   MACRO     MSGADR
        MOV       AH,09H
        MOV       DX,OFFSET MSGADR
        INT       21H

        XOR       BH,BH
        CMP       CMDBUF[BX-2],'/'
        JNE       ERROR
        MOV       DX,OFFSET CMDBUF[1]
        MOV       CMDBUF[BX-2],0

        MOV       AX,4300H
        INT       21H
        JC        ERROR
        TEST      CX,00011000B
        JNE       ERROR
GETSW
        CMP       AL,':'
        JE        ?SWITCH
        AND       AL,NOT('A' XOR 'a')
        CMP       AL,'R'
```

指定されたファイルの属性を調べ、その値をCXレジスタにセットするファンクションリクエスト

ここでキャリーフラグが「1」であれば、上のファンクションリクエストは成功していない。したがって、このようにキャリーフラグを調べて、それが「1」であれば、エラーメッセージ表示ルーチンへジャンプさせる。このような場合の多くは、指定したファイルが存在しないか、あるいはそのパス名に誤りがある

ここでは指定されたファイルの属性がCXレジスタに正しく得られている

図 4.8　ファンクションリクエスト実行時のエラー検出の例

図4.8は、指定したファイルの属性を知るためのファンクションリクエストの部分です。このファンクションリクエストにより、そのファイルの属性(属性の種別を表すコード)がCXレジスタにセットされますが、もし指定したファイルが存在していない場合は、CXレジスタの値は意味を持たないことになります。ですから、コールから戻ったら、CXレジスタの値を使う前に、まず指定したファイルが存在していたか否かのチェックを行う必要があります。

指定したファイルが存在していたか否かは、コールから戻った直後のキャリーフラグの状態によって知ることができます。キャリーフラグが0の場合は、目的のファイルの属性が正常に得られており(つまり指定したファイルが存在していた)、1の場合は目的のファイルが存在せず、CXレジスタの値は無意味であり、使ってはいけないことを示しています(**図 4.9** に示すファンクション 43_H の入出力条件にあるように、この時点のAXレジスタに、エラー種別を表すコードがセットされているが、この例でのエラーは通常 0003_H の「指定されたパス名が存在しない」のみである)。

このように、キャリーフラグによるファンクションリクエスト実行時のエラー情報を利用すること

により、ファンクションリクエストが正常に実行されなかった場合の対処が可能になります。1章での応用例はその最も簡単なもので、キャリーフラグが1の場合は、エラー種別を調べることなく、すべて「Illegal switch or filename」のエラーメッセージを表示するようなプログラムになっています。

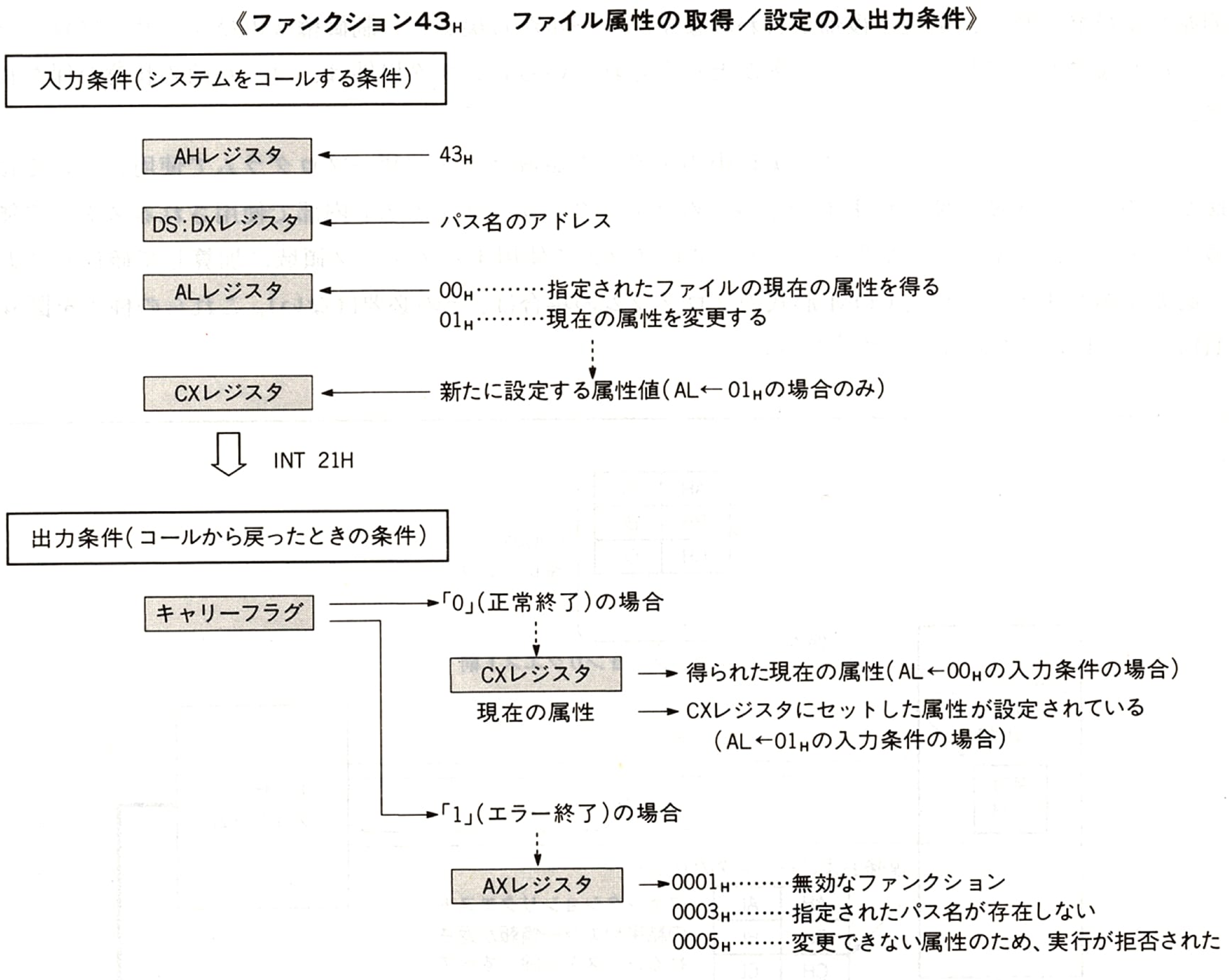

《ファンクション43H　ファイル属性の取得／設定の入出力条件》
入力条件(システムをコールする条件)
AHレジスタ
43H
DS:DXレジスタ
パス名のアドレス
ALレジスタ
00H………指定されたファイルの現在の属性を得る
01H………現在の属性を変更する
CXレジスタ
新たに設定する属性値(AL← 01Hの場合のみ)
INT 21H
出力条件(コールから戻ったときの条件)
キャリーフラグ
「0」(正常終了)の場合
CXレジスタ
得られた現在の属性(AL←00Hの入力条件の場合)
現在の属性
CXレジスタにセットした属性が設定されている
(AL←01Hの入力条件の場合)
「1」(エラー終了)の場合
AXレジスタ
0001H………無効なファンクション
0003H………指定されたパス名が存在しない
0005H………変更できない属性のため、実行が拒否された

図 4.9　ファンクション 43Hの入出力条件

■ ファンクションリクエストによるほかのレジスタおよびスタックポインタへの影響

ファンクションリクエストの実行前と実行後に、CPUの各レジスタの値はどのような変化するのでしょうか。結論からいえば、それぞれのファンクションリクエストの出力条件に使用されるレジスタ、それにエラーコードがセットされるAXレジスタを除いて(つまり、そのファンクションの結果に関係するレジスタを除いて)すべてのレジスタはコール直前の状態が保存されており、影響を受けません。

ファンクションリクエストの実行は、INT 21Hのソフトウェア割り込みによって、プログラムの制御権がユーザープログラムからOS側に移り、コールされたファンクションがシステムによって実行されます。その実行には、当然のことながら各種のレジスタが使用されますが、それは私たちが見ることのできないブラックボックス内で行われます。ただし、ファンクションリクエスト内部で使用されるすべてのレジスタは、コール直前の状態を保存するために、ブラックボックス内のプログラムの開始および終了時に、退避と復帰が行われます。コールから戻ると、制御権は再びユーザープログラムに戻りますが、結果やエラーコードがセットされているレジスタ以外は、コール前の状態が保たれているのです。

また、ファンクションリクエスト実行中のスタック領域は、ユーザープログラムで使用している領域がそのまま引き続き使われます。そのためファンクションリクエスト内部で使用されるスタック領域の分として、128バイト程度をユーザープログラムで使用するスタック領域に加算して確保しておく必要があります（ただし、COM形式のプログラムの場合は、その必要はない）。これらの様子を**図4.10**に示しますので参考にしてください。

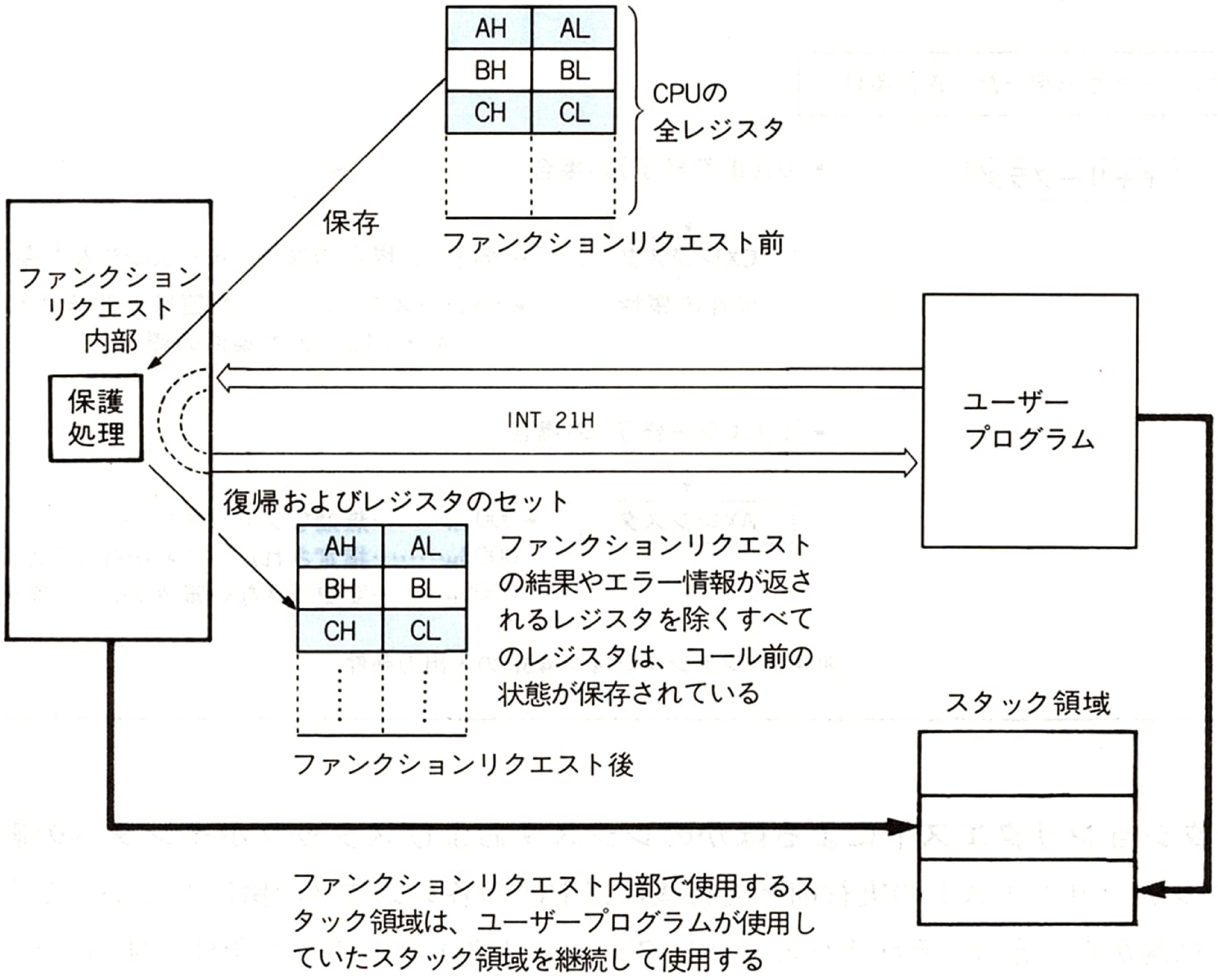

AH
AL
BH
BL
CH
CL
CPUの
全レジスタ
ファンクションリクエスト前
保存
ファンクション
リクエスト
内部
保護
処理
INT 21H
ユーザー
プログラム
復帰およびレジスタのセット
AH
AL
BH
BL
CH
CL
ファンクションリクエスト
の結果やエラー情報が返さ
れるレジスタを除くすべて
のレジスタは、コール前の
状態が保存されている
ファンクションリクエスト後
スタック領域
ファンクションリクエスト内部で使用するス
タック領域は、ユーザープログラムが使用し
ていたスタック領域を継続して使用する

図4.10　ファンクションリクエスト実行前後の各レジスタと、使用されるスタック領域

4.3 システムコール

MS-DOS のシステムコールは、通常 INT 21H によるファンクションリクエストの機能以外にも、以下に示す INT 20H から INT 27H までのシステムコールによる機能がユーザーに提供されています。つまり、INT 21H(割り込みタイプ 21H)は、いくつかのシステムコールによる機能の 1 つなのです。

INT 20H ［プログラムの終了］

現在のプログラム(プロセス)を終了して親プロセスに戻る。この機能は、ファンクションリクエストのファンクション 00_H とまったく同じであり、通常のプログラムの場合は、ユーザープログラムを終了して COMMAND.COM に戻ることになる。このシステムコールは、旧バージョンとの互換性のために残っているが、現在ではファンクションリクエストの $4C_H$ を使うことが望ましい。

［入力条件］	CS レジスタ	←	プログラムセグメント・プレフィックスのセグメントアドレス(ただし、COM 形式の場合は、もともとこのアドレスがセットされているので、この操作は必要ない)
［出力条件］	なし		

INT 21H ［ファンクションのリクエスト］

それぞれのファンクションに必要なパラメータを準備したあと、特定の機能の実行を要求する。このシステムコールを、ファンクションリクエストと呼ぶ。コール条件、リターン条件は、要求するそれぞれのファンクションによる。

INT 25H ［アブソリュートディスクリード］

MS-DOS のファイルシステムを介さず、ディスクの指定セクタを直接読み出す。この操作は MS-DOS システムの管理外であり、利用には十分な注意が必要である。

[入力条件]　ALレジスタ　←　ドライブ番号(A：＝00H、B：＝01H、……)
DS：BXレジスタ　←　読み出したデータの先頭アドレス
CXレジスタ　←　一度に読み出すセクタ数
DXレジスタ　←　読み出す最初のセクタの論理セクタ番号(2章の2.3.4参照)
[出力条件]　キャリーフラグ　＝　1……エラーが発生した
＝　0……正常終了

INT　26H　[アブソリュートディスクライト]

MS-DOSのファイルシステムを介さず、ディスクの指定セクタに直接書き込む。これは、INT 25Hと逆の機能(読み出しが書き込みに)である。この操作もMS-DOSシステムの管理外であり、利用には十分な注意が必要である。

[入力条件]　ALレジスタ　←　ドライブ番号(A：＝00H、B：＝01H、…)
DS：BXレジスタ　←　書き込むデータの先頭アドレス
CXレジスタ　←　一度に書き込むセクタ数
DXレジスタ　←　書き込む最初のセクタの論理セクタ番号(2章の2.3.4参照)
[出力条件]　キャリーフラグ　＝　1……エラーが発生した
＝　0……正常終了

INT　27H　[プログラムをメモリに置いたまま終了]

現在のプログラム(プロセス)をメモリ上に置いたまま終了し、親プロセスに戻る。つまり、それまで使用していた、ユーザープログラムの占有領域を解放しない。EXE形式の場合は、通常この割り込みは使用しない。このシステムコールも、旧バージョンとの互換性のために残っているが、現在では、ファンクション31Hを利用すべきである。

[入力条件]　CS：DXレジスタ　←　メモリに置いたまま終了するプログラムコードの最終バイトの次のアドレス。ただし、使用する場合はCOM形式なのでCSレジスタを操作する必要がない。
[出力条件]　なし

次のものは、システムの機能を呼び出すためのソフトウェア割り込みではなく、システム内でエラーなどが発生したときに、システムが自動的に呼び出すものである。したがって、ユーザープログラムでそれらの割り込みベクタテーブルに独自の処理ルーチンのアドレスを格納することにより、自分で用意した処理ルーチンが呼び出されるようにすることもできる。

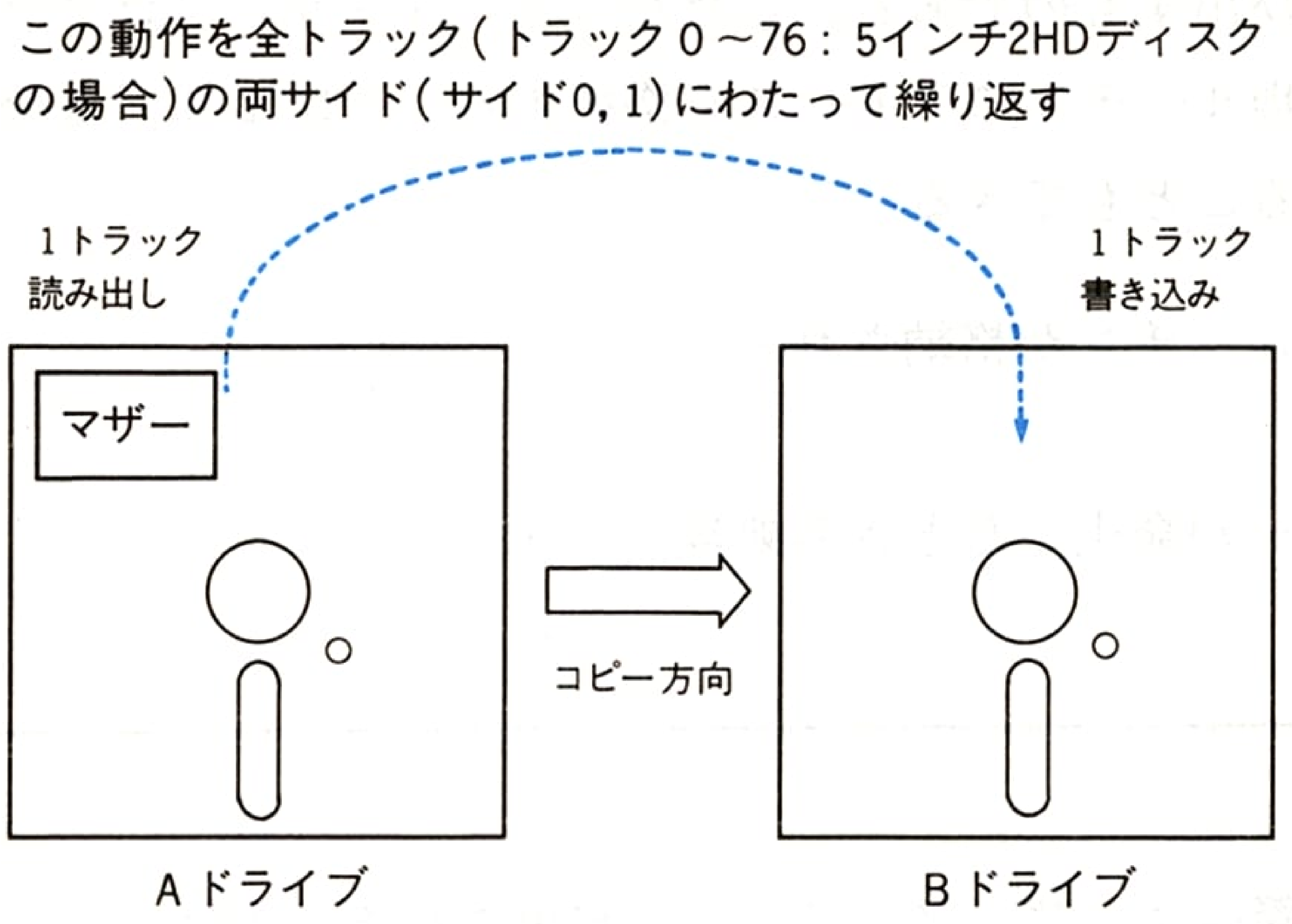
- **INT 22H** …… プログラムの終了処理ルーチンが格納される
- **INT 23H** …… Ctrl-C の処理ルーチン
- **INT 24H** …… 致命的なディスクエラーが発生したときの処理ルーチン

システムコールによるファンクションで、実際にユーザープログラムで利用されるものは、ファンクションリクエストで使う21_Hのほかに、20_Hの「プログラムの終了」と、ディスク to ディスクのコピープログラムや、ディスクの物理的ダンププログラムなどに使う25_Hの「アブソリュートディスクリード」および26_Hの「アブソリュートディスクライト」です。

20_Hのプログラムの終了の割り込みは、ファンクションリクエストのファンクションナンバー00_Hのプログラムの終了とまったく同じ機能であり、COM 形式のプログラムを終了する手段として使われていましたが、この20_Hの割り込みの機能は、ファンクションリクエストのファンクション$4C_H$の「プロセスの終了」として、さらに機能を拡張して提供されていますので、通常はファンクション$4C_H$のファンクションリクエストを使います。また、27_Hの「プログラムをメモリに置いたまま終了」も、ファンクションリクエストのファンクション31_Hの「キーププロセス」として、さらに機能を拡張して提供されていますので、これもそちらを使います。

■ システムコールを使った応用プログラム例

ユーザープログラムで実際に使われるシステムコールの機能については、さきほど述べましたが、ここでは、INT 25H と 26H のアブソリュートディスクリード／ライトの機能を利用して、ディスクコピープログラムを作成してみましょう。このプログラムは、MS-DOS のシステムディスクに含まれているディスクコピープログラム DISKCOPY と同じ働きをするものです。ただし、プログラムを簡略にするため、ディスクの種類は、8 インチ 2D または 5 インチおよび 3.5 インチ 2HD に限定し、コピー方向を、A ドライブから B ドライブにコピーされるように固定します(**図 4.11**)。このプログラム名を「DCOPY」としましょう。


この動作を全トラック（トラック0～76：5インチ2HDディスクの場合）の両サイド（サイド0, 1）にわたって繰り返す
1トラック読み出し
1トラック書き込み
マザー
コピー方向
Aドライブ
Bドライブ

このプログラム名を「DCOPY」とし、

A＞DCOPY ⏎

によってDCOPYプログラムを起動してから、送り側（マザー）ディスクをAドライブにセットし、受け側ディスクをBにセットする。何らかのキー入力によりコピーが開始される

図 4.11　ディスク to ディスクのコピープログラム DCOPY の機能

リスト 4.1 に、DCOPY プログラムのソースファイルを示します。このプログラムには、本節の主題である、システムコールによる各種のファンクションの応用をはじめ、INT 21H によるファンクションリクエストの応用なども含まれていますので、その使い分けに注目してください。

```
 ;; DCOPY.ASM     DISK COPY
 ;;
 ;;       02H     Display Character
 ;;       09H     Display String
 ;;       0CH     Flush buff/Read Keyboard
 ;;       0DH     Disk Reset
 ;;       3BH     Change Current Directory
 ;;       INT 25H Absolute Disk Read
 ;;       INT 26H Absolute Disk Write

NTRACK  EQU     77 ……………………………………………トラック数
NSIDE   EQU     2 ……………………………………………サイド数
NSECTOR EQU     8 ……………………………………………セクタ数
SECSIZE EQU     1024 …………………………………………セクタサイズ
DSKSIZE EQU     NTRACK*NSIDE*NSECTOR*SECSIZE ……ディスク容量
BUFSIZE EQU     NSIDE*NSECTOR*SECSIZE ……………バッファサイズ(一度に読み込むデータのバイト数)
NUM     EQU     NTRACK   ;DSKSIZE/BUFSIZE ………転送回数

CSEG    SEGMENT ………………論理セグメントCSEGの始まり
        ASSUME  CS:CSEG,DS:CSEG ……………セグメントレジスタはCSEGにセットされているとする

        ORG     100H ……………COM形式の場合は必ず0100Hとする
```

このプログラムは5インチ2HD用であるがこの値を変えれば他のディスクにも使える。このプログラムでは1トラックずつ転送している

— リスト 4.1 — （次ページ以下に続く）

```
START:  MOV    AH,09H              ····文字列出力のファンクションリクエスト ┐
        MOV    DX,OFFSET STRTMES       {AH=09H                          ├ スタートメッセージを出力する    ┐
        INT    21H                     {DS:DX=文字列の先頭アドレス      ┘                                 │
        MOV    AX,0C08H            ····入力バッファをクリアしてキーボード入 ┐ 任意のキーが押されることを開   │
        INT    21H                     力するファンクションリクエスト   │ 始の合図とする                   │
                                       {AH=0CH                          │ 先行入力があると「危険」なので  │ 初
                                       {AL=目的のキー入力ファンクションナンバー ┘ それをクリアするコールを用い │ 期
                                                                          る(入力した文字は捨てる)       │ 設
        MOV    AH,09H              ····文字列出力のファンクションリクエスト ┐                                 │ 定
        MOV    DX,OFFSET MESCR         {AH=09H                          ├ 改行する（カーソルを次の行の    │
        INT    21H                     {DS:DX=文字列の先頭アドレス      ┘ 先頭におくための)               ┘
        MOV    CX,NUM ····転送を繰り返す回数

LOP:    PUSH   CX ············残り回数を保存する
        MOV    AH,02H              ·············1文字出力のファンクションリクエスト ┐ 1トラック転送する      ┐
        MOV    DL,'*'                  {AH=02H                          ├ 「*」マークを1個表         │
        INT    21H                     {DL=出力する文字                 ┘ 示していく                 │
        MOV    AL,0                ·············アブソリュート・ディスクリードのシステムコール ┐          │
        MOV    BX,OFFSET BUFFER        {AL=ドライブ番号                 │ Aドライブ                  │
        MOV    CX,BUFSIZE/SECSIZE      {DS:BX=ディスク転送アドレス      ├ から1トラ                  │
        MOV    DX,CURNUM               {CX=読み出しセクタ数             │ ック読み出す               │
        INT    25H                     {読み出し開始論理セクタ番号      ┘                            │
        JC     RERROR ·············キャリーフラグが「1」ならエラー                                       │
        POPF ·············このコールではフラグがスタックに積まれたままなので取り出す                     │
        MOV    AL,1                ·············アブソリュート・ディスクライトのシステムコール ┐         │
        MOV    BX,OFFSET BUFFER        {AL=ドライブ番号                 │ Bドライブに               │ 転
        MOV    CX,BUFSIZE/SECSIZE      {DS:BX=ディスク転送アドレス      ├ 1トラック書               │ 送
        MOV    DX,CURNUM               {CX=書き込みたいセクタ数         │ き込む                    │ 処
        INT    26H                     {DX=書き込み論理セクタ番号       ┘                            │ 理
        JC     WERROR ·············キャリーフラグが「1」ならエラー                                       │
        POPF ·············このコールではフラグがスタックに積まれたままなので取り出す                     │
        ADD    CURNUM,BUFSIZE/SECSIZE ·············現在の論理セクタ番号を進める                          │
        POP    CX ·············残り回数を復帰                                                             │
        LOOP   LOP ·············残り回数が0になるまで繰り返す                                             │
                                                                                                         │
        MOV    AH,09H              ·············文字列出力のファンクションリクエスト ┐                   │
        MOV    DX,OFFSET ENDMES        {AH=09H                          ├ エンドメッセージを          │
        INT    21H                     {DS:DX=文字の先頭アドレス        ┘ 出力する                    ┘
        JMP    SHORT RETURN

RERROR: MOV    DX,OFFSET RERRMES ·············読み出し時のエラーメッセージのアドレスをDXレジスタにセットする
        JMP    SHORT MESOUT

WERROR: MOV    DX,OFFSET WERRMES ·············書き込み時のエラーメッセージのアドレスをDXレジスタにセットする
MESOUT: MOV    AH,09H              ·············文字列出力のファンクションリクエスト ┐
        INT    21H                     {AH=09H                          ├ エラーメッセージの出力
                                       {DS:DX=文字列の先頭アドレス      ┘
        MOV    AL,1
        MOV    ERRCOD,AL
```

1トラックずつコピー

```
RETURN: MOV     AH,0DH          ……ディスクリセットのファンクションリクエスト
        INT     21H                   AH=0DH
        MOV     AH,3BH          ……カレントディレクトリ変更の
        MOV     DX,OFFSET ROOTEA      ファンクションリクエスト
        INT     21H                   {AH=3BH
        MOV     AH,3BH                {DS：DX=パス名のアドレス
        MOV     DX,OFFSET ROOTEB
        INT     21H
        MOV     AL,ERRCOD
        MOV     AH,4CH          ……プロセス終了のファンクションリクエスト
        INT     21H                   {AH=4CH
                                      {AL=終了コード

CURNUM  DW      0 ……………現在の論理セクタ番号を格納しておく
BUFFER  DB      BUFSIZE DUP (?) ……………ディスク転送用バッファ(1トラック分)
STRTMES DB      0DH,0AH,"I will copy from A: to B:",0DH,0AH } スタートメッセージ
        DB      "          Hit any key           ",'$'
MESCR   DB      0DH,0AH,'$' ……………改行用文字列
ENDMES  DB      0DH,0AH,"...Completed",0DH,0AH,'$' ……………エンドメッセージ
RERRMES DB      0DH,0AH,"Read Error",0DH,0AH,'$' ……………リードエラーメッセージ
WERRMES DB      0DH,0AH,"Write Error",0DH,0AH,'$' ……………ライトエラーメッセージ
ROOTEA  DB      "A:¥",0 } ディレクトリをルートに戻すためのパス名
ROOTEB  DB      "B:¥",0
ERRCOD  DB      0

CSEG     ENDS ……………論理セグメントCSEGの終わり
         END    START ……………プログラム開始アドレスを指定する
```

このプログラムを終了する際に、システムを介さないディスクアクセスを行っているのでディスクバッファをクリアし、カレントディレクトリをルートに戻しておく

注）INT 25H、INT 26Hではセグメントレジスタ以外のレジスタはすべて破壊されるので、必要ならば退避する

リスト 4.1　ディスクコピープログラム DCOPY のソースプログラム

このDCOPYプログラムを作成する手順は、1章で作成したCHMODプログラムの作成の場合と同じです(1章1.3参照)。まずエディタを使って、ソースファイルをファイル名「DCOPY.ASM」として作成したあと、アセンブラMASMを実行し、リンカLINK、さらにEXE2BINを実行することにより、実行可能なDCOPYプログラム「DCOPY.COM」が完成します。

```
A>MASM DCOPY,,DCOPY;↵ ……………ソースファイルDCOPY.ASMをアセンブルする
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


  46680 + 349029 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors ……………アセンブル終了。アセンブルエラーなし

A>LINK DCOPY;↵ ……………アセンブルによって生成されたDCOPY.OBJファイルに
                        対してリンカを実行する

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

LINK : warning L4021: no stack segment ……………COMファイルを作成する場合、この警告は無視してよい

A>EXE2BIN DCOPY DCOPY.COM↵ ……………リンクによって生成されたEXEファイルを、COMファイルに変換する

A>
              以上の作業で実行可能なDCOPY.COMが作成されている
```

図 4.12　DCOPY プログラム作成のためのアセンブル作業例

以上の作業で、実行可能なディスクコピープログラム「DCOPY.COM」が完成しました。では、このプログラムを実行して、何か適当なディスクをコピーしてみましょう。その実行例を図 **4.13** に示します。A ドライブに、オリジナルディスクとして MS-DOS のシステムディスクをセットして、そのバックアップコピーを作る例です。B ドライブにセットするディスクは、当然のことながら、MS-DOS でフォーマット処理済みのものでなくてはなりません。

```
A>DCOPY↵ ……………DCOPYプログラムを起動する

I WILL COPY FROM A: TO B: ｝このオープニングメッセージが表示されたら、Aドライブにマザー、
       HIT ANY KEY        ｝Bドライブに受け側ディスクをセットして何らかのキーを押す
*****************************************************************************
...COMPLETED ……………全トラックのコピーが終了すると、
                        完了メッセージが表示される
                                          └─コピー動作が開始されると、1トラックが
A>                                          コピーされるごとに「*」が1つ表示される。
                                            よって、全部で77個(5インチ2HDの場合)
                                            の「*」が表示されるはずである
```

図 4.13　完成した DCOPY プログラムの実行例

4.4 主要ファンクションリクエスト実例集

ここでは、ファンクションリクエストをユーザープログラムで利用するための基礎知識を、いくつかの代表的なファンクションの基本的なプログラミング例をもとに解説しましょう。以前にも少し触れましたが、MS-DOSのバージョン$2.x$や$3.x$のファンクションリクエストにも、バージョン1.25時代の古いファンクションリクエストがサポートされていますが、古いファンクションには、ファイルの階層構造がサポートされないなど、いろいろな支障がありますので、特別の目的以外には使用しない方がよいでしょう。このような、現在では普通使われない古いファンクションと、それに代わるファンクションとの対照表を**表4.2**に示します。

旧ファンクション ⇨		現在はこちらを使用する	
バージョン1.25時代のファンクションの呼び名とその番号		左に代わるバージョン2.x以降のファンクションの呼び名とその番号	
00_H	プログラムの終了	$4C_H$	プロセスの終了
$0F_H$	ファイルのオープン	$3D_H$	ハンドルのオープン
10_H	ファイルのクローズ	$3E_H$	ハンドルのクローズ
11_H	最初のディレクトリエントリのサーチ	$4E_H$	最初のファイルのサーチ
12_H	次のディレクトリエントリのサーチ	$4F_H$	次のファイルのサーチ
13_H	ファイルの削除	41_H	ディレクトリエントリの削除
14_H	シーケンシャルリード	$3F_H$	リードハンドル
15_H	シーケンシャルライト	40_H	ライトハンドル
16_H	ファイルの生成	$3C_H$	ハンドルの生成
17_H	ファイル名の変更	56_H	ディレクトリエントリの変更
21_H	ランダムリード	$3F_H$	リードハンドル
22_H	ランダムライト	40_H	ライトハンドル
23_H	ファイルサイズを得る*	42_H	ファイルポインタの移動
24_H	相対レコードのセット	42_H	ファイルポインタの移動
26_H	新しいPSPの生成	$4B00_H$	プログラムのロードと実行
27_H	ランダム・ブロックリード	$3F_H$	リードハンドル
28_H	ランダム・ブロックライト	40_H	ライトハンドル

＊通常はファイルを終わりからアクセスする場合に使う

注）それぞれのファンクションの呼び名はそれぞれのファンクションの「タイトル名」にすぎず、その内容の違いを表しているわけではない。

表4.2　旧／新 ファンクション対照表

■ ファンクション 05_H、$0A_H$、$4C_H$ を利用したプログラム例
（バッファードキーボード入力とプリンタへの出力プログラム BUFFPRN）

［使用ファンクション］

ファンクション 05_H ……… プリンタへの出力
ファンクション $0A_H$ ……… バッファードキーボード入力
ファンクション $4C_H$ ……… プログラムの終了

このサンプルプログラムは、キーボードから適当な文字列を入力してそれをバッファリングしておき、リターンキーを入力すると、入力した文字列がプリンタに出力されるものです。40文字（プログラム中で任意設定）を超える入力の場合は、最初の40文字がプリンタに出力されます。この場合、40文字を超える入力にはベルが鳴りますので確認してください。また、このファンクション $0A_H$ による入力時には、DEL キー、BS キーおよびテンプレート機能などによる入力訂正編集機能が働きますので、ミスタイプをした場合など、これらの機能で正しい文字列に整えてからプリントアウトできます。なお、このバッファードキーボード入力の仕組みについては**図 4.14** を参照してください。

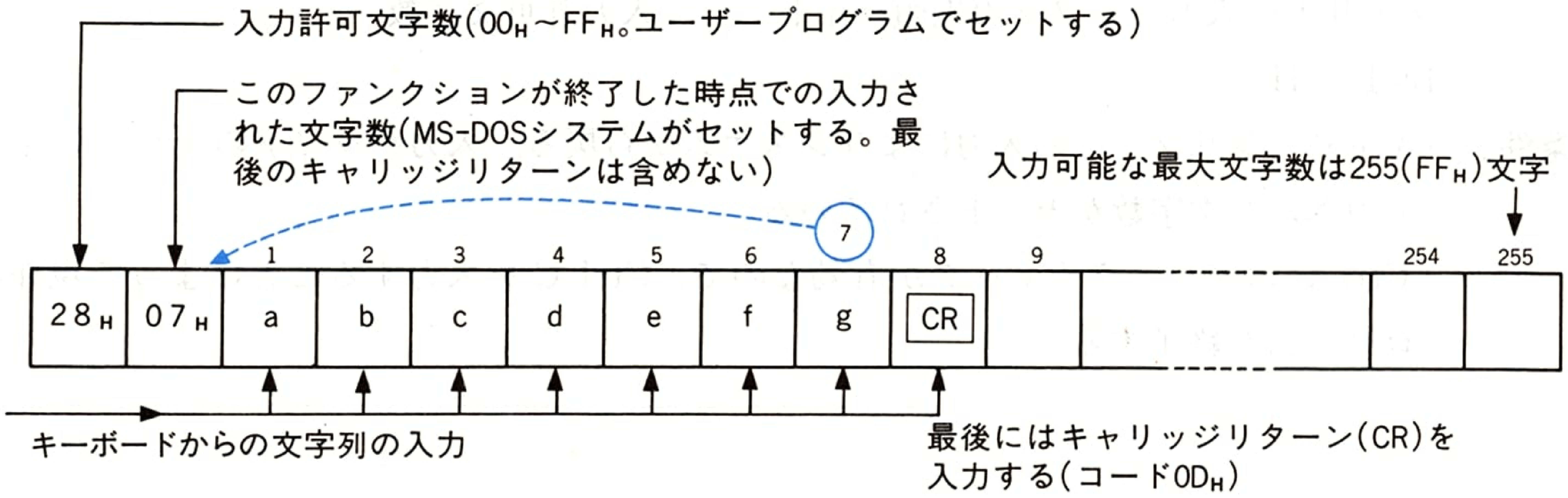

入力許可文字数（00_H～FF_H。ユーザープログラムでセットする）
このファンクションが終了した時点での入力された文字数（MS-DOSシステムがセットする。最後のキャリッジリターンは含めない）
入力可能な最大文字数は255（FF_H）文字
7
1
2
3
4
5
6
8
9
254
255
28_H
07_H
a
b
c
d
e
f
g
CR
キーボードからの文字列の入力
最後にはキャリッジリターン（CR）を入力する（コード$0D_H$）

図 4.14　入力バッファの動作

ところで、多くのドットシリアルプリンタは、ラインフィード（LF）またはキャリジリターン（CR）が入力されるか、あるいはプリンタ内部の入力バッファがフルになるまで印字動作をしませんので、プリンタを扱うプログラムでは注意が必要です。このプログラムの場合は、バッファリングされていた文字列の最後は必ずキャリジリターンなので、そのたびに印字します。なお、ページプリンタは、改ページコード（^L）が送られないと印字しませんので、その点を考慮する必要があります。

ファンクション $0A_H$ には、Ctrl-C によるブレークチェック機能がありますので、Ctrl-C を入力することでもプログラムは終了します。ここで使用するファンクションの入出力条件を以下に示します。

ファンクション 05H ［プリンタへの1文字出力］

［入力条件］ AHレジスタ　←　05H
DLレジスタ　←　出力する文字コード
INT 21H

［出力条件］
- 目的の文字がプリンタ(PRNデバイス)に出力される
- Ctrl-Cのブレークチェックが有効なので、Ctrl-Cを入力することによって現在のプログラムが終了する

ファンクション 0AH ［バッファードキーボード入力］

［入力条件］ AHレジスタ　←　0AH
DS：DXレジスタ　←　セグメントアドレス
- 入力バッファの先頭アドレス。COM形式のプログラムであれば、DSレジスタの操作は必要なし

メモリ上の入力バッファの先頭バイト　←　入力許可文字数
INT 21H

［出力条件］
- キャリッジリターンの入力によりシステムから戻る。入力バッファの2バイト目には、入力された文字数がセットされている
- Ctrl-Cのブレークチェックが有効なので、Ctrl-Cを入力することによって現在のプログラムが終了する

ファンクション 4CH ［プログラムの終了］

［入力条件］ AHレジスタ　←　4CH
ALレジスタ　←　プログラムの終了コード。通常は0を、エラーのあったときは0以外を入れる
INT 21H

［出力条件］ 現在実行中のプログラム(プロセス)が終了し、制御権が親プロセスに戻る。親プロセスは、子プロセスの返した終了コードをファンクション4DHを用いて受け取ることができる。また、バッチ実行中では、IF　ERRORLEVELで終了コードを調べることができる

ではこのプログラム名を「BUFFPRN」(BUFFered PRiNt)として、その処理の流れを**図 4.15** に、そのソースファイルを**リスト 4.2** に示します。

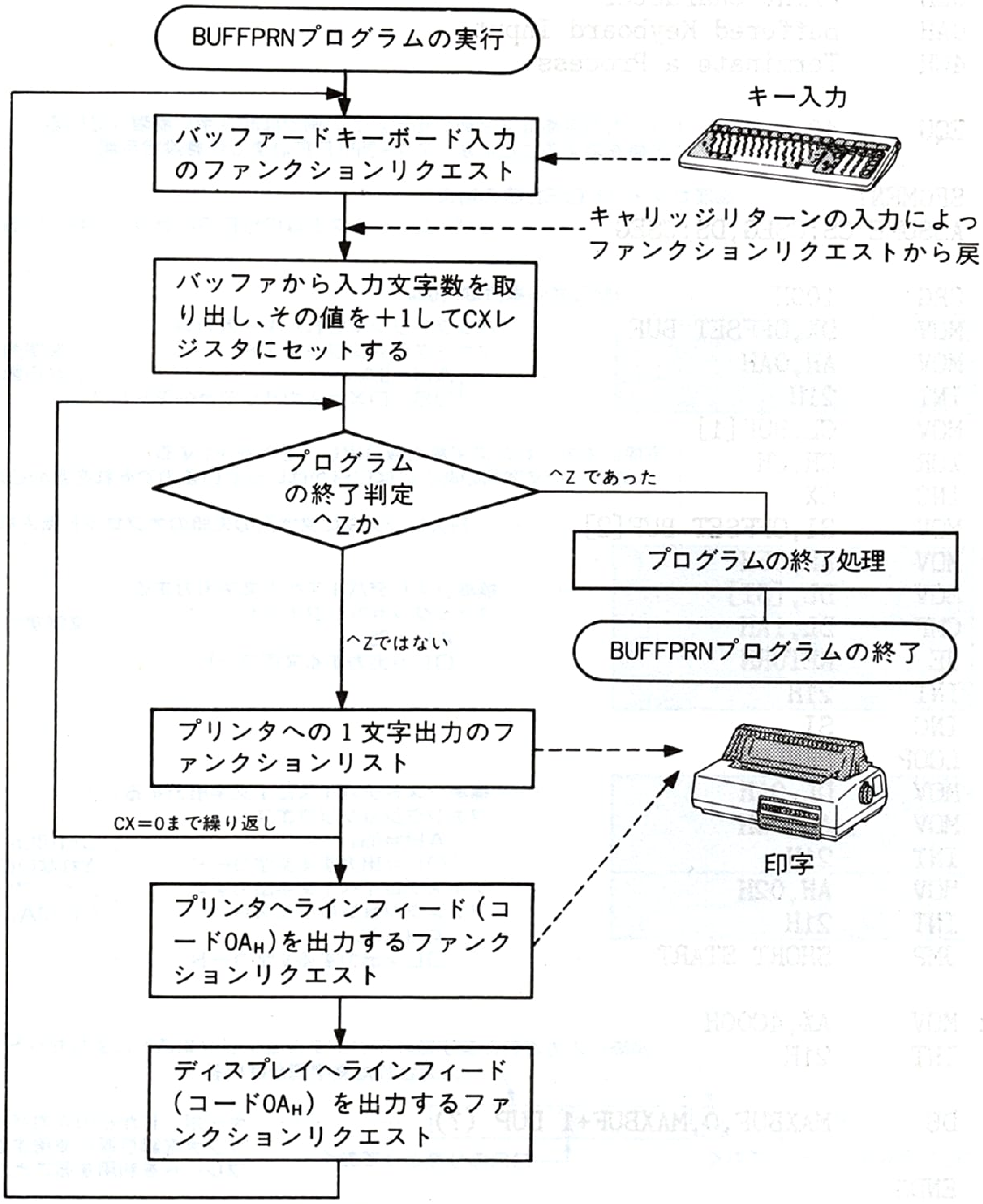

BUFFPRNプログラムの実行
バッファードキーボード入力のファンクションリクエスト
キー入力
キャリッジリターンの入力によってファンクションリクエストから戻る
バッファから入力文字数を取り出し、その値を＋1してCXレジスタにセットする
プログラムの終了判定 ^Zか
^Z であった
プログラムの終了処理
BUFFPRNプログラムの終了
^Zではない
プリンタへの 1 文字出力のファンクションリスト
印字
CX=0まで繰り返し
プリンタへラインフィード（コードOAH）を出力するファンクションリクエスト
ディスプレイへラインフィード（コードOAH）を出力するファンクションリクエスト

図 4.15　BUFFPRN プログラムの処理の流れ

```
;; BUFFPRN.ASM  Bufferd Keyboard Input
;;               and Print Out
;;
;;       02H     Display Character
;;       05H     Print Character
;;       0AH     Buffered Keyboard Input
;;       4CH     Terminate a Process

MAXBUF  EQU      40

CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG

        ORG      100H
START:  MOV      DX,OFFSET BUF
        MOV      AH,0AH
        INT      21H
        MOV      CL,BUF[1]
        XOR      CH,CH
        INC      CX
        MOV      SI,OFFSET BUF[2]
PRNLOP: MOV      AH,05H
        MOV      DL,[SI]
        CMP      DL,1AH
        JE       RETURN
        INT      21H
        INC      SI
        LOOP     PRNLOP
        MOV      DL,0AH
        MOV      AH,05H
        INT      21H
        MOV      AH,02H
        INT      21H
        JMP      SHORT START

RETURN: MOV      AX,4C00H
        INT      21H

BUF     DB       MAXBUF,0,MAXBUF+1 DUP (?)
CSEG    ENDS
        END      START
```

MAXBUF EQU 40 ……1行に入力可能な文字数の指定。この例では40文字に制限している この値を変えることによって0〜255(FF_H)まで任意設定可能

CSEG SEGMENT ……論理セグメントCSEGの始まり

ASSUME CS:CSEG,DS:CSEG ……セグメントレジスタはCSEGにセットされているとする

ORG 100H ……COM形式の場合は0100_H

START: MOV DX,OFFSET BUF / MOV AH,0AH / INT 21H ……バッファリングされたキー入力のファンクションリクエスト
{ $AH = 0A_H$
{ DS : DX＝入力バッファのアドレス
} 文字列をキーボードから読み込む

MOV CL,BUF[1] / XOR CH,CH / INC CX } 実際に入力された文字数を取り出し、それを+1する（入力された文字列のあとにはCRがはいっているのでそれを含めるため）

MOV SI,OFFSET BUF[2] ……BXレジスタに文字列の先頭のオフセットを入れる

PRNLOP: MOV AH,05H / MOV DL,[SI] / CMP DL,1AH / JE RETURN / INT 21H ……標準リストデバイスへ1文字出力するファンクションリクエスト
{ $AH = 05_H$
{ DL＝出力する文字コード
} 1文字ずつプリンタに出力する

MOV DL,0AH / MOV AH,05H / INT 21H ……標準リストデバイスに1文字出力するファンクションリクエスト
{ $AH = 05_H$
{ DL＝出力する文字コード

MOV AH,02H / INT 21H ……ディスプレイへ1文字出力するファンクションリクエスト
{ $AH = 02_H$
{ DL＝出力する文字コード

} CR($0D_H$)だけでは改行されないので、ディスプレイ、プリンタ双方にLF($0A_H$)を出力する

BUF DB MAXBUF,0,MAXBUF+1 DUP (?)
- MAXBUF ← 1行の入力許可文字数をセットしておく
- 0 ← 実際に入力された文字数がファンクションリクエストによりセットされる
- MAXBUF+1 DUP (?) ← 入力された文字列がはいる
- +1 ← CRの分をとっておく
- ……キーボードからの入力バッファ。同じ入力バッファを繰り返し使用することによってテンプレートを利用することができる

END START ……プログラム開始アドレスを指定する

リスト 4.2 ファンクション 05_H、$0A_H$ のサンプルプログラム BUFFPRN のソースプログラム

このソースプログラムから実行可能なオブジェクトプログラム BUFFPRN.COM を作成し、それを実行してみましょう。実行例を図 4.16 に示します。

キーボードから直接入力した場合

```
A>BUFFPRN↵
These characters are printed on paper.↵
^Z↵
A>
```

BUFFPRNプログラムを実行する

文字を入力し、各行の最後はリターンキーを入力する

プログラムを終了するにはCtrl-Zを入力する

各行の最後のリターンキーの入力と同時に、その行がプリンタに印字される

プリンタにはこのように出力される

```
These characters are printed on paper.
```

ファイルをリダイレクトで入力した場合

```
A>COPY CON/A TEXT↵
These characters are printed on paper.↵
^Z↵
        1 個のファイルをコピーしました.

A>BUFFPRN <TEXT↵
These characters are printed on paper.
^Z^C

A>
```

実験用のファイルTEXTを作成する

ファイルTEXTを、リダイレクトで入力する

ファイルTEXTの内容がディスプレイにも表示されてしまう

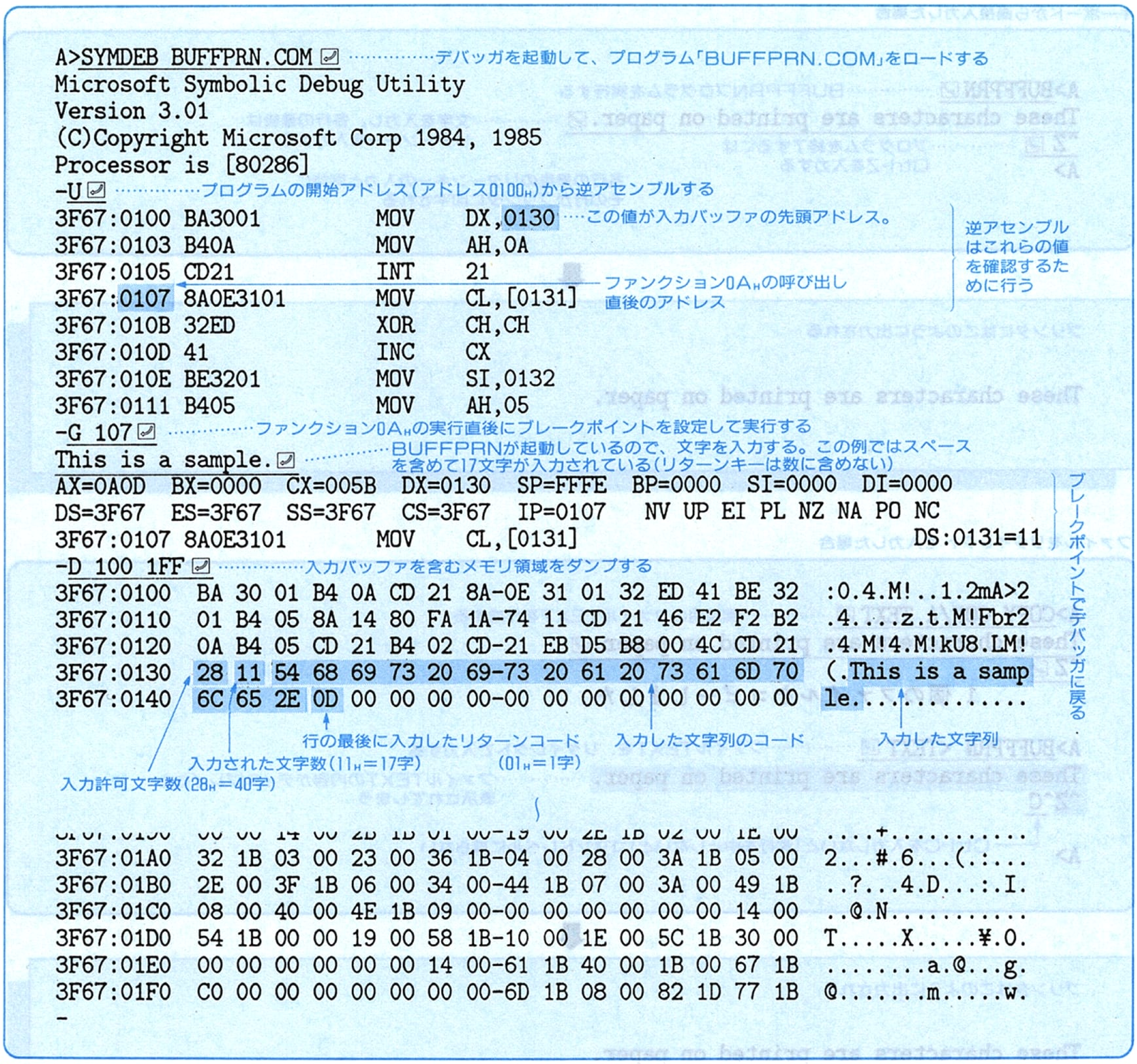
Ctrl-Cを入力しないと(実行を中止しないと)コマンドレベルに戻らない

プリンタはこのように出力される

```
These characters are printed on paper.
```

図 4.16　ファンクション 05_{H}、$0A_{\mathrm{H}}$ のサンプルプログラム BUFFPRN の実行例

次に、ファンクション $0A_{\mathrm{H}}$ の実行後の入力バッファのメモリ内容を見てみましょう。そのために、今回はデバッガ SYMDEB(あるいは DEBUG)を起動したあとに、サンプルプログラムの BUFFPRN を、ブレークポイントを設定して実行し、ダンプして確認します。**図 4.17** にその実行例を示します。



```
A>SYMDEB BUFFPRN.COM↵
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-U↵
3F67:0100 BA3001           MOV    DX,0130
3F67:0103 B40A             MOV    AH,0A
3F67:0105 CD21             INT    21
3F67:0107 8A0E3101         MOV    CL,[0131]
3F67:010B 32ED             XOR    CH,CH
3F67:010D 41               INC    CX
3F67:010E BE3201           MOV    SI,0132
3F67:0111 B405             MOV    AH,05
-G 107↵
This is a sample.↵
AX=0A0D  BX=0000  CX=005B  DX=0130  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=3F67  ES=3F67  SS=3F67  CS=3F67  IP=0107   NV UP EI PL NZ NA PO NC
3F67:0107 8A0E3101         MOV    CL,[0131]                    DS:0131=11
-D 100 1FF↵
3F67:0100  BA 30 01 B4 0A CD 21 8A-0E 31 01 32 ED 41 BE 32  :0.4.M!..1.2mA>2
3F67:0110  01 B4 05 8A 14 80 FA 1A-74 11 CD 21 46 E2 F2 B2  .4...z.t.M!Fbr2
3F67:0120  0A B4 05 CD 21 B4 02 CD-21 EB D5 B8 00 4C CD 21  .4.M!4.M!kU8.LM!
3F67:0130  28 11 54 68 69 73 20 69-73 20 61 20 73 61 6D 70  (.This is a samp
3F67:0140  6C 65 2E 0D 00 00 00 00-00 00 00 00 00 00 00 00  le..............

3F67:01A0  32 1B 03 00 23 00 36 1B-04 00 28 00 3A 1B 05 00  2...#.6...(.:...
3F67:01B0  2E 00 3F 1B 06 00 34 00-44 1B 07 00 3A 00 49 1B  ..?...4.D...:.I.
3F67:01C0  08 00 40 00 4E 1B 09 00-00 00 00 00 00 00 14 00  ..@.N...........
3F67:01D0  54 1B 00 00 19 00 58 1B-10 00 1E 00 5C 1B 30 00  T.....X.....¥.0.
3F67:01E0  00 00 00 00 00 00 14 00-61 1B 40 00 1B 00 67 1B  ........a.@...g.
3F67:01F0  C0 00 00 00 00 00 00 00-6D 1B 08 00 82 1D 77 1B  @.......m.....w.
-
```

デバッガを起動して、プログラム「BUFFPRN.COM」をロードする

プログラムの開始アドレス（アドレス0100_{H}）から逆アセンブルする

この値が入力バッファの先頭アドレス。

ファンクション$0A_{H}$の呼び出し直後のアドレス

逆アセンブルはこれらの値を確認するために行う

ファンクション$0A_{H}$の実行直後にブレークポイントを設定して実行する

BUFFPRNが起動しているので、文字を入力する。この例ではスペースを含めて17文字が入力されている（リターンキーは数に含めない）

ブレークポイントでデバッガに戻る

入力バッファを含むメモリ領域をダンプする

行の最後に入力したリターンコード（01_{H}＝1字）

入力した文字列のコード

入力した文字列

入力された文字数（11_{H}＝17字）

入力許可文字数（28_{H}＝40字）

図 4.17　BUFFPRN プログラムの実行例（デバッガを起動ておいて実行した場合）

ダンプ結果に示した、入力バッファの1バイト目の最大入力文字数、2バイト目の実際に入力された文字数、3バイト目以降の入力された文字列に注目して、ファンクション 0AH の機能を確認してください。

■ ファンクション $3D_H$、$3F_H$、$3E_H$、02_H を利用したプログラム例
(2章のファイルの読み出しプログラムFREAD)

[使用ファンクション]

ファンクション $3D_H$ ………ハンドル(ファイル)のオープン
ファンクション $3F_H$ ………ハンドル(ファイル)の読み出し
ファンクション $3E_H$ ………ハンドル(ファイル)のクローズ
ファンクション 02_H ………ディスプレイへの1文字出力

このサンプルプログラムは、2章で作成したファイルの読み出しプログラムFREADそのものです。忘れてしまった方は、もう一度2章を読み直しておいてください。ここではそのFREADプログラムのファンクションリクエストを中心に、プログラム全体を詳しく解説しましょう。

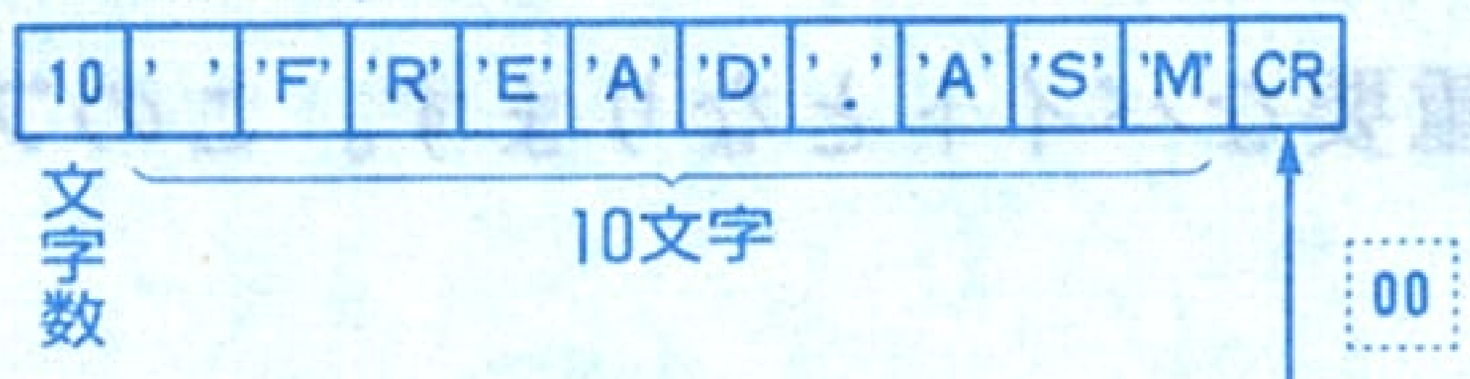
FREADプログラムは、テキストファイルの内容をディスプレイに表示するもので、内蔵コマンドであるTYPEに相当します。ここでは、このプログラムを、そのソースファイル上で細かく解説していきましょう(**リスト4.3**)。なお、このプログラムの流れについては、2章の図2.1を参照してください。

```
;; FREAD.ASM       File Read Program
;;
;;         02H     Display Character
;;         09H     Display String
;;         3DH     Open a File
;;         3EH     Close a File Handle
;;         3FH     Read From File/Device
;;         4CH     Terminate a Process


CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG
        ORG     80H
CMDLEN  DB      ?
CMDBUF  DB      127 DUP (?)

        ORG     100H
START:  MOV     BL,CMDLEN
        CMP     BL,2
        JB      NERROR
        XOR     BH,BH
        MOV     CMDBUF[BX],0
        MOV     DX,OFFSET CMDBUF[1]

        MOV     AX,3D00H
        INT     21H
```

SEGMENT ……………論理セグメントCSEGの始まり

ASSUME CS:CSEG,DS:CSEG,ES:CSEG …………セグメントレジスタはCSEGにセットされているとする

ORG 80H ～ CMDBUF DB 127 DUP (?)：CSEGのオフセット0080_Hからはコマンドラインのコマンド名を除いたものがはいっている
例）A>FREAD FREAD.ASM ⏎ なら…



ORG 100H ……………COM形式の場合は0100_H

MOV BL,CMDLEN ～ JB NERROR：コマンドラインの文字数が0、つまりパラメータがなければエラー

XOR BH,BH ～ MOV CMDBUF[BX],0：ファイル名(パス名)の末尾の次に00_Hを書き込み、ASCIZ文字列にする

MOV DX,OFFSET CMDBUF[1] ～ INT 21H …………ファイルオープンのファンクションリクエスト
AH=$3D_H$
DS：DX=パス名のアドレス
AL=アクセスモード

— リスト4.3 —（次ページに続く）

```
         JC      NERROR  ……………キャリーフラグが「1」ならエラー（オープンできない）
         MOV     BX,AX  ……………AXレジスタにファイルハンドルが返される。
                                 BXレジスタに保存し、以下のループで値を保持する

LOP:     MOV     AH,3FH            ……………ファイル読み出しのファンクションリクエスト
         MOV     DX,OFFSET BUF          {AH=3FH
         MOV     CX,1                   {DS:DX=バッファアドレス
         INT     21H                    {CX=読み出すバイト数
         TEST    AX,AX  }
         JZ      RDEND  } 実際に読み込まれたバイト数がAXレジスタに返される。0なら終了
         MOV     DL,BUF }
         CMP     DL,1AH } Ctrl-Z(1AH)を読み込んだら終了する
         JE      RDEND  }
         MOV     AH,02H            ……………ディスプレイへ文字を出力するファンクションリクエスト
         INT     21H                    {AH=02H
         JMP     SHORT LOP              {DL=出力する文字コード

RDEND:   MOV     AH,3EH            ……………ファイルクローズのファンクションリクエスト
         INT     21H                    {AH=3EH
         XOR     AL,AL                  {BX=ファイルハンドル
         JMP     SHORT RETURN

NERROR:  MOV     DX,OFFSET NERRMES ……………ディスプレイへ文字列を出力するファンク } エラーメッセージ
         MOV     AH,09H                  ションリクエスト                    } の出力
         INT     21H                    {AH=09H                              }
         MOV     AL,1                   {DS:DX=文字列の先頭アドレス          }
RETURN:  MOV     AH,4CH            ……………プロセス終了のファンクションリクエスト
         INT     21H                    {AH=4CH
                                        {AL=終了コード
         このバッファを介して、ファイルから読み出されたデータがディスプレイに出力される
BUF      DB      0 ……………ファイルからの入力バッファ
NERRMES  DB      0DH,0AH,"Not found",0DH,0AH,'$' ……………エラーメッセージ
CSEG     ENDS ……………論理セグメントCSEGの終わり
         END     START ……………プログラム開始アドレスを指定する（COM形式なので先頭）
```

リスト 4.3　ファンクション $3D_H$、$3F_H$、$3E_H$、02_H のサンプルプログラム（2章の FREAD プログラム）

ファンクション $3D_H$ の「ハンドルのオープン」において、AL レジスタにセットする1バイトのコードは、ファイルアクセス・コントロールと呼ばれる情報であり、子プロセスやファイル・シェアリング（1つのファイルを同時に複数のプログラムからアクセスすること）を実行するアプリケーションなどでは重要なバイトとなります。この**ファイルアクセス・コントロール**の概要を**図 4.18** に示します。

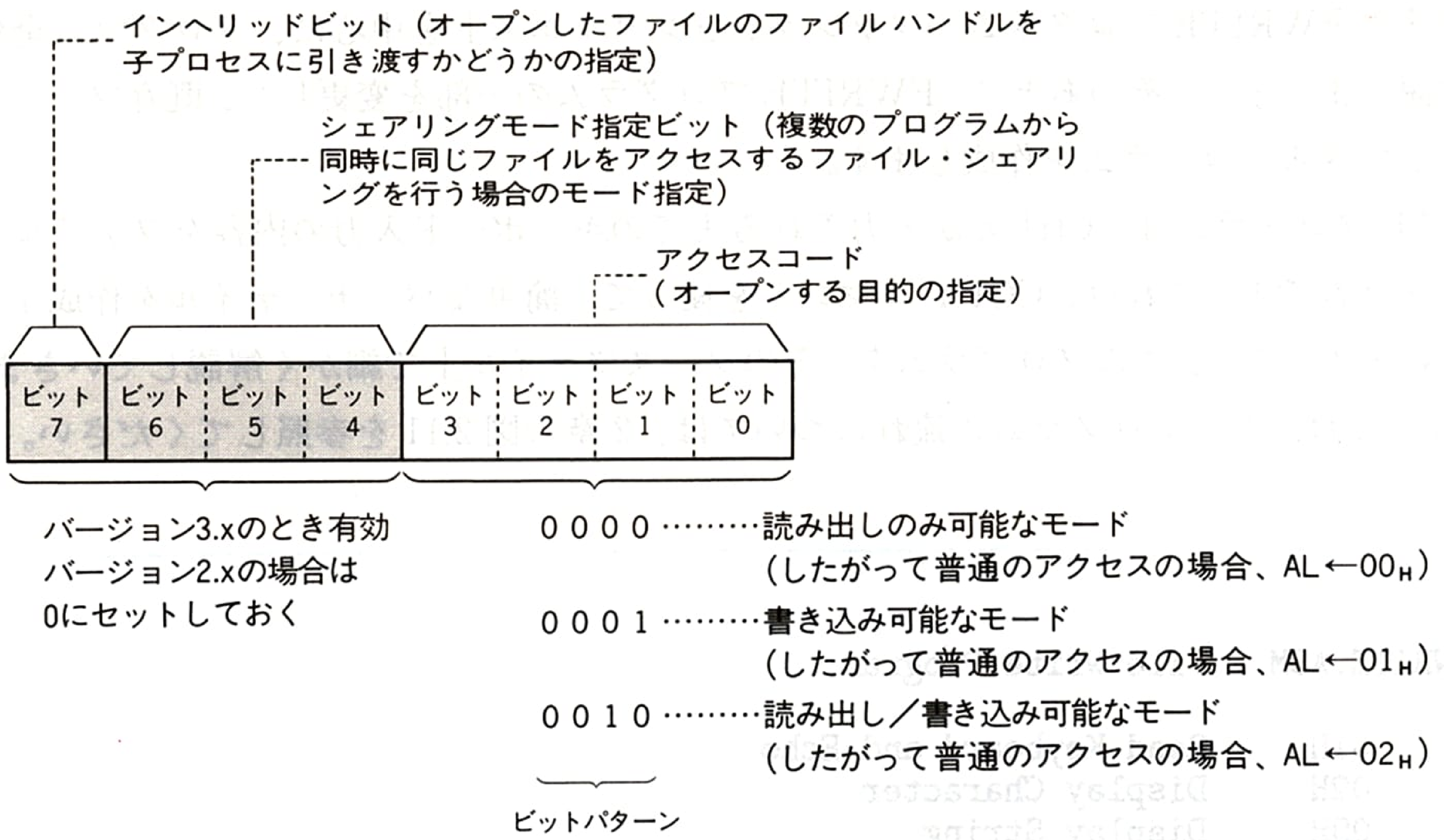

インヘリッドビット（オープンしたファイルのファイルハンドルを
子プロセスに引き渡すかどうかの指定）
シェアリングモード指定ビット（複数のプログラムから
同時に同じファイルをアクセスするファイル・シェアリ
ングを行う場合のモード指定）
アクセスコード
（オープンする目的の指定）
ビット 7
ビット 6
ビット 5
ビット 4
ビット 3
ビット 2
ビット 1
ビット 0
バージョン3.xのとき有効
バージョン2.xの場合は
0にセットしておく
0 0 0 0………読み出しのみ可能なモード
（したがって普通のアクセスの場合、AL←00_H）
0 0 0 1………書き込み可能なモード
（したがって普通のアクセスの場合、AL←01_H）
0 0 1 0………読み出し／書き込み可能なモード
（したがって普通のアクセスの場合、AL←02_H）
ビットパターン

図 4.18　ファイルアクセス・コントロールバイトの概要

ただし、子プロセスなどを実行しない普通のアプリケーションでは、ファイルアクセス・コントロールは、上位 4 ビットを 0 にして、アクセスコードのみをセットします。つまり、次の 3 つの場合をセットするだけでよいことになります。

00_H：　ファイルの読み出しが目的でオープンする場合
AL レジスタ　←　01_H：　ファイルの書き込みが目的でオープンする場合
02_H：　ファイルの読み出しと書き込みが目的でオープンする場合

そのため、FREAD プログラムでは、ソースプログラムのように、AL レジスタには 00_H をセットしています。

■ ファンクション 3C$_H$、40$_H$、3E$_H$、42$_H$ を利用したプログラム例（2 章のファイルへの書き込みプログラム FWRITE）

［使用ファンクション］

ファンクション 3C$_H$ ………ハンドル（ファイル）の生成
ファンクション 40$_H$ ………ハンドル（ファイル）の書き込み
ファンクション 3E$_H$ ………ハンドル（ファイル）のクローズ
ファンクション 42$_H$ ………ファイルポインタの移動

このサンプルプログラムは、2章で作成したファイルの新規作成プログラムFWRITEと同じです。ここではそのFWRITEプログラムのファンクションリクエストを中心に、プログラム全体について詳しく解説しましょう。そのあとで、FWRITEプログラムの一部を変更して、既存ファイルへの追加書き込みができるプログラムを作成します。

FWRITEプログラムは、Ctrl-Zが入力されるまでのキーボード入力の内容をファイルとして作成するプログラムです。これは、COPYコマンドを使って、簡単なバッチファイルを作成する場合の働きに似ています。では、このプログラムを、そのソースファイル上で細かく解説していきましょう（**リスト4.4**）。なお、このプログラムの流れについては、2章の図2.11を参照してください。

```
;; FWRITE.ASM   File Write Program
;;
;;      01H     Read Keyboard and Echo
;;      02H     Display Character
;;      09H     Display String
;;      3CH     Create a File
;;      3EH     Close a File Handle
;;      40H     Write to File/Device
;;      4CH     Terminate a Process

CSEG    SEGMENT ……………論理セグメントCSEGの始まり
        ASSUME CS:CSEG,DS:CSEG ……………セグメントレジスタはCSEGにセットされているとする
        ORG     80H           ｝CSEGのオフセット0080Hからにはコマンドラインの
CMDLEN  DB      ?             ｝コマンド名を除いたものがはいっている
CMDBUF  DB      127 DUP (?)   ｝例）A>FWRITE TEST↵ なら…… | 5 | ' ' | 'T' | 'E' | 'S' | 'T' | CR |
                                                        文字数    5文字           ↑ 00

        ORG     100H ……………COM形式の場合は0100H
START:  MOV     BL,CMDLEN     ｝コマンドラインの文字数が0、つまり
        CMP     BL,2          ｝パラメータなしならエラーとする
        JB      CERROR        ｝
        XOR     BH,BH         ｝パラメータ、つまり書き込むファイル名の末尾の次に00Hを書き込み
        MOV     CMDBUF[BX],0  ｝ASCIZ文字列にする
        MOV     DX,OFFSET CMDBUF[1] ……………ファイル（ハンドル）クリエイトのファンクションリクエスト
        MOV     CX,0                    ｛AH=3CH
        MOV     AH,3CH                  ｛DS：DX=パス名のASCIZ文字列のアドレス
        INT     21H                     ｛          （DSレジスタはすでにそこを指している）
                                        ｛CX=ファイル属性（通常のファイルは0）
        JC      CERROR ……………キャリーフラグが「1」ならエラー（オープンできない）
        MOV     BX,AX ……………ファイルハンドルがAXレジスタに返される
                               （以下のループではBXレジスタは破壊されない）
```

— リスト4.4 —（次ページに続く）

```
LOP:     MOV     AH,01H          ……キーボードから1文字入力するファンクションリクエスト
         INT     21H                   AH=01H
         MOV     BUF,AL          ……入力された文字がALレジスタに返されるので、それを出力バッファに入れる
         MOV     DX,OFFSET BUF   ……ファイル(ハンドル)への書き込みのファンクションリクエスト
         MOV     CX,1                 {AH=40H
         MOV     AH,40H               {DS:DX=出力バッファの先頭アドレス
         INT     21H                  {CX=書き込むバイト数
                                      {BX=ファイルハンドル
         CMP     BUF,1AH  }
         JE      WRTEND   } 入力した文字がCtrl-Z(1AH)であれば終了
         CMP     BUF,0DH
         JNE     LOP
         MOV     DL,0AH          ……ディスプレイへの1文字出力の        }
         MOV     AH,02H            ファンクションリクエスト             } 入力した文字がCR
         INT     21H                  {AH=02H                          } (0DH)であればディス
                                      {DL=出力する文字                  } プレイおよび出力フ
         MOV     BUF,0AH                                               } ァイルにLF(0AH)
         MOV     DX,OFFSET BUF   ……ファイル(ハンドル)への書き込みの    } を追加する。
         MOV     AH,40H            ファンクションリクエスト             } (MS-DOSの標準
         MOV     CX,1                 {AH=40H                          } テキスト形式に合わ
         INT     21H                  {DS:DX=出力バッファの先頭アドレス  } せる)
         JMP     SHORT LOP            {CX=書き込みバイト数              }
                                      {BX=ファイルハンドル              }

WRTEND:  MOV     AH,3EH          ……ファイル(ハンドル)クローズの }
         INT     21H               ファンクションリクエスト      } 正常終了へ
         XOR     AL,AL                {AH=3EH                   }
         JMP     SHORT RETURN         {BX=ファイルハンドル        }

CERROR:  MOV     DX,OFFSET CERRMES ……文字列出力のファンクションリクエスト }
         MOV     AH,09H               {AH=09H                            } エラーメッセージの出力
         INT     21H                  {DS:DX=文字列の先頭アドレス          }
         MOV     AL,1
RETURN:  MOV     AH,4CH          ……プロセス終了のファンクションリクエスト
         INT     21H                  {AH=4CH
                                      {AL=終了コード

BUF      DB      0 ……ファイルへの出力バッファ
CERRMES  DB      0DH,0AH,"Can't create",0DH,0AH,'$' ……エラーメッセージ

CSEG     ENDS ……論理セグメントCSEGの終わり
         END     START ……プログラム開始アドレスを指定する
```

このバッファを介してキー入力がファイルへ書き込まれる

リスト 4.4　ファンクション 3CH、40H、3EH のサンプルプログラム(2 章の FWRITE プログラム)

さて、この FWRITE プログラムは、ファイルを新規作成するプログラムでしたが、この一部を変更すれば、既存ファイルに追加書き込みをするプログラムができます。このプログラム名を「FADD」(File ADD)としておきましょう。さきの FWRITE プログラムと次の FADD プログラムとの機能の違いを図 4.19 に示します。

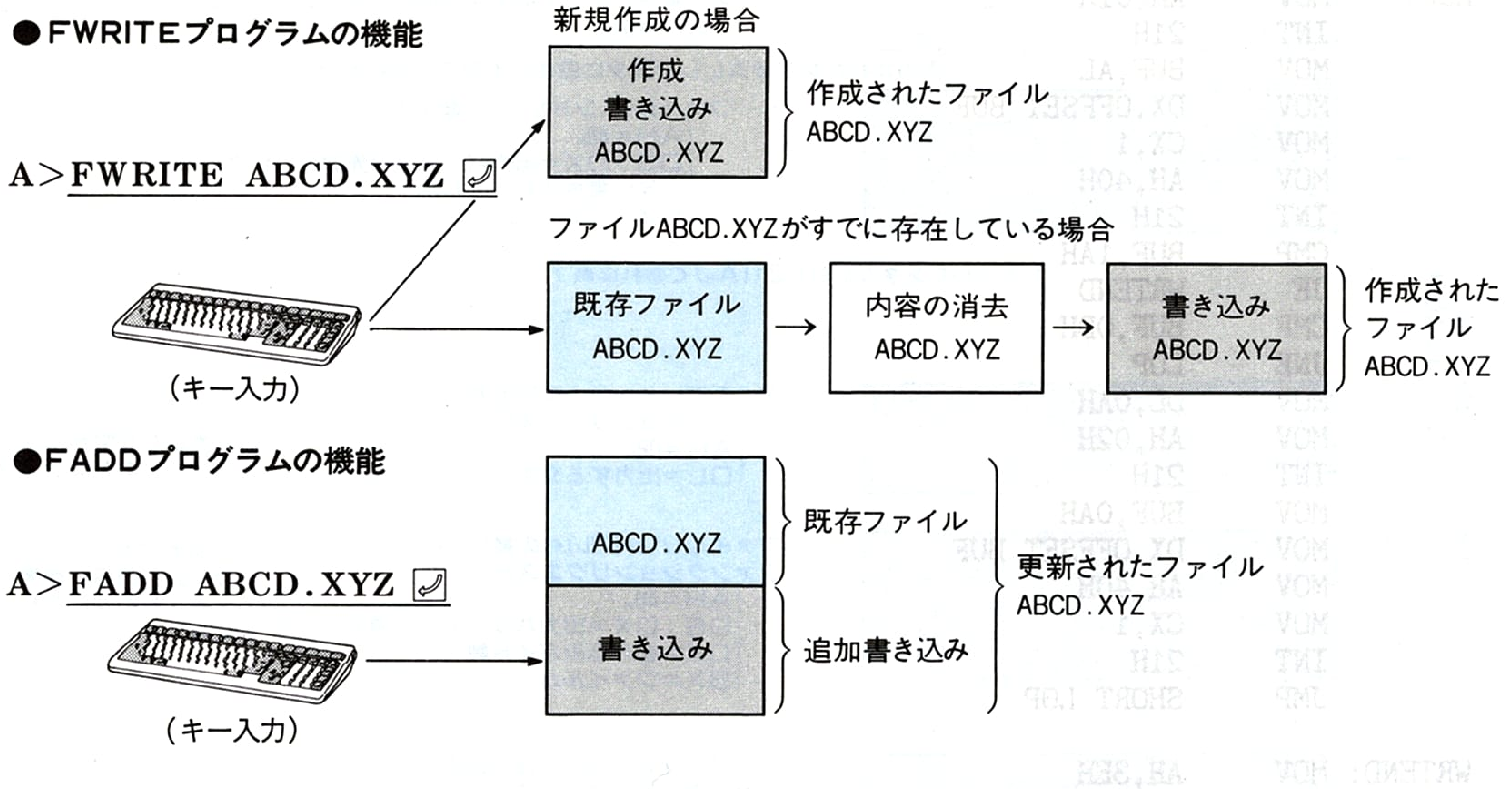

●FWRITEプログラムの機能
A>FWRITE ABCD.XYZ
（キー入力）
新規作成の場合
作成
書き込み
ABCD.XYZ
作成されたファイル
ABCD.XYZ
ファイルABCD.XYZがすでに存在している場合
既存ファイル
ABCD.XYZ
内容の消去
ABCD.XYZ
書き込み
ABCD.XYZ
作成された
ファイル
ABCD.XYZ
●FADDプログラムの機能
A>FADD ABCD.XYZ
（キー入力）
ABCD.XYZ
既存ファイル
書き込み
追加書き込み
更新されたファイル
ABCD.XYZ

図 4.19　FADD プログラムと FWRITE プログラムとの機能の違い

ところで、MS-DOS のテキストファイル形式では、ファイルの末尾に EOF コード（Ctrl-Z、つまり $1A_H$）を付けることになっています。これは、ファイルを 128 バイトのブロック単位でしか扱えなかった CP／M 時代の名残であり、ファイルのサイズをバイト単位で管理できる MS-DOS では、とくに必要があるわけではありません。過去との互換性のため、EOF コードを付加するツール（エディタなど）もありますが、最近では EOF コードを付けないツールも多くなっています。

このプログラムでは、ファイルの末尾を読み込んで、EOF が付いているかどうかを判定することにしましょう。（図 4.20 参照）

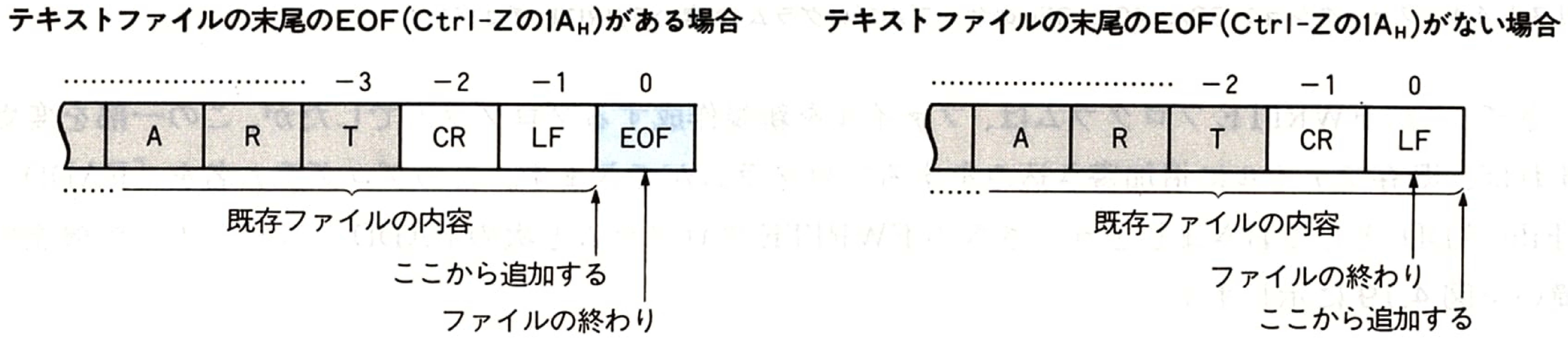

テキストファイルの末尾のEOF（Ctrl-Zの1AH）がある場合
−3
−2
−1
0
A
R
T
CR
LF
EOF
既存ファイルの内容
ここから追加する
ファイルの終わり
テキストファイルの末尾のEOF（Ctrl-Zの1AH）がない場合
−2
−1
0
A
R
T
CR
LF
既存ファイルの内容
ファイルの終わり
ここから追加する

図 4.20　MS-DOS のテキストファイルの末尾の形式

では次に、既存ファイルの最後に追加書き込みをするFADDプログラムのソースファイルを示して解説しましょう。（**リスト4.5**）

```
;; FADD.ASM     File Append Program
;;
;;        01H     Read Keyboard and Echo
;;        02H     Display Character
;;        09H     Display String
;;        3DH     Open a File
;;        3EH     Close a File Handle
;;        3FH     Read From File/Device
;;        40H     Write to File/Device
;;        42H     Move a File Pointer
;;        4CH     Terminate a Process

CSEG      SEGMENT
          ASSUME CS:CSEG,DS:CSEG
          ORG     80H
CMDLEN    DB      ?
CMDBUF    DB      127 DUP (?)

          ORG     100H
START:    MOV     BL,CMDLEN
          CMP     BL,2
          JB      CERROR
          XOR     BH,BH
          MOV     CMDBUF[BX],0
          MOV     DX,OFFSET CMDBUF[1]
          MOV     AX,3D02H
          INT     21H
          JC      CERROR
          MOV     BX,AX
          MOV     CX,-1
          MOV     DX,CX
          MOV     AX,4202H
          INT     21H
          MOV     DX,OFFSET BUF
          MOV     CX,1
          MOV     AH,3FH
          INT     21H
          JC      LOP
          CMP     BUF,1AH
          JNE     LOP
          MOV     CX,-1
          MOV     DX,CX
          MOV     AX,4201H
          INT     21H
```

CX DX ＝－1

ファイルポインタの移動

- 移動の始点
  AL＝00_Hファイルの先頭から
  01_H現在の位置から
  02_Hファイルの終わりから
- 移動するバイト数
  | CX | DX |
  の32ビットの符号付整数で表す

▭の部分がFWRITE.ASMを変更した箇所。その他はFWRITEのソースプログラムを参照

ファイル（ハンドル）オープンのファンクションリクエスト
AH＝$3D_H$
AL＝アクセスモード01_Hは書き込みモードの指定
DS：DX＝オープンするファイルのパス名のアドレス

ファイルポインタの移動のファンクションリクエスト
AH＝42_H
AL＝移動方法
CX：DX＝移動するバイト数
BX＝ファイルハンドル

ファイル読み出しのファンクションリクエスト
AH＝$3F_H$
DS：DX＝バッファアドレス
CX＝読み出すバイト数

― リスト4.5 ―（次ページに続く）

```
LOP:    MOV     AH,01H
        INT     21H
        MOV     BUF,AL
        MOV     DX,OFFSET BUF
        MOV     CX,1
        MOV     AH,40H
        INT     21H
        CMP     BUF,1AH
        JE      WRTEND
        CMP     BUF,0DH
        JNE     LOP
        MOV     DL,0AH
        MOV     AH,02H
        INT     21H
        MOV     BUF,0AH
        MOV     DX,OFFSET BUF
        MOV     AH,40H
        MOV     CX,1
        INT     21H
        JMP     SHORT LOP

WRTEND: MOV     AH,3EH
        INT     21H
        XOR     AL,AL
        JMP     SHORT RETURN

CERROR: MOV     DX,OFFSET CERRMES
        MOV     AH,09H
        INT     21H
        MOV     AL,1
RETURN: MOV     AH,4CH
        INT     21H

BUF     DB      0
CERRMES DB      0DH,0AH,"Can't open",0DH,0AH,'$'

CSEG    ENDS
        END     START
```

リスト 4.5　FADD プログラムのソースプログラム（2 章の FWRITE プログラムにファンクション 42_H を追加）

完成した実行可能なプログラム「FADD.COM」の実行例を図 **4.21** に示します。

```
A>TYPE TEST↵ ……………既存ファイル「TEST」の内容を表示する
ABCDEFGHIJ  ┐
1234567890  ┘その内容

A>FADD TEST↵ ……………既存ファイル「TEST」に対してFADDプログラムを実行する
File append test.↵ ……………追加する文字列を入力する
^Z ……………Ctrl-Zの入力によりFADDプログラムを終了する（PC-9800シリーズでは画面はクリアされる）

A>TYPE TEST↵ ……………実行結果を表示して確認する
ABCDEFGHIJ
1234567890
File append test. ……………入力した行が追加されている

A>
```

図 4.21　FADD プログラムの実行例

■ ファンクション 40H を利用したプログラム例(標準出力を利用した FREAD プログラム)

[使用ファンクション]

ファンクション 40H ……… ハンドルの書き込み

これまでディスプレイへの出力にはファンクション 02H や 09H など、ディスプレイへの出力用のファンクションを利用しました。これらのファンクションは MS-DOS バージョン 1.25 から存在するもので、標準エラー出力に出力できないとか、ファンクション 09H では文字「$」が文字列の末尾を示すために使われているので文字「$」そのものを出力することができないなど、あまり洗練された仕様とはいえません。手軽さとわかりやすさゆえに現在でも用いられることが多いのですが、のちほど解説するように今後はこれらのファンクションの代わりにファイルハンドルを利用した方法を使うことが望ましいでしょう。

2.1 節や 3.6.2 項で解説したように、ファイルハンドルの 0 番から 4 番まではシステムによって予約されています。これらのファイルハンドルはシステムによってあらかじめオープンされているので、いきなり読み出しや書き込みを行うことができます。このことを利用して、コンソールへの入出力をファイルへの入出力とまったく同じ方法で行うことができます。

次のプログラムではファイルハンドルの 1 番と 2 番、つまり標準出力と標準エラー出力を利用したディスプレイへの出力例を示します。具体的には、ファイルから読み込んだデータをそのままファイルハンドルの 1 番に出力することによってファイルの内容をディスプレイに表示し、ファイルハンドルの 2 番に文字列を出力することによってエラーメッセージを表示します。プログラムの流れについては、**図 4.22** を参照してください。プログラム名は「STDFREAD」としましょう(**リスト 4.6**)。

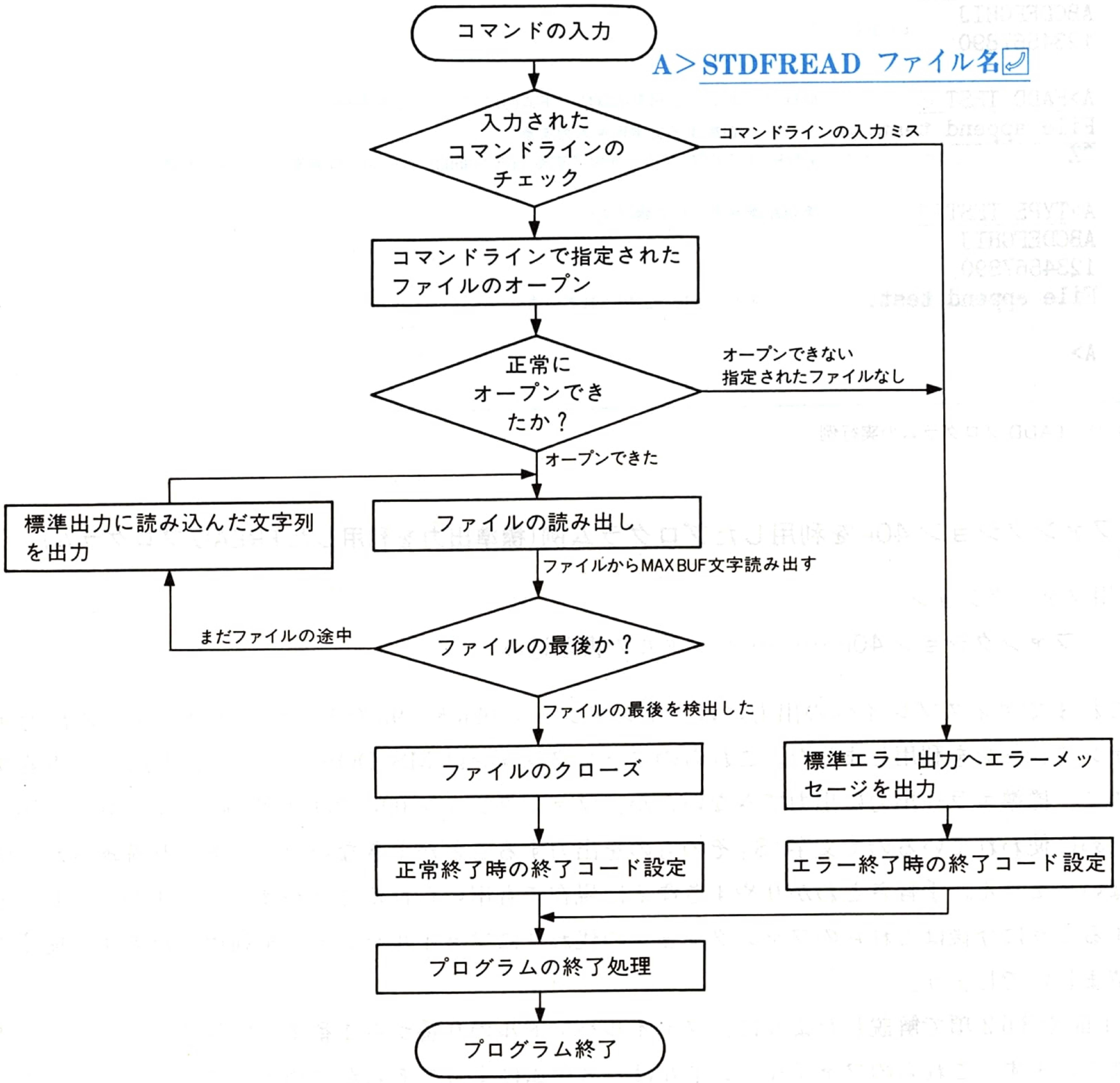

コマンドの入力
A＞STDFREAD ファイル名⏎
入力されたコマンドラインのチェック
コマンドラインの入力ミス
コマンドラインで指定されたファイルのオープン
正常にオープンできたか？
オープンできない
指定されたファイルなし
オープンできた
ファイルの読み出し
標準出力に読み込んだ文字列を出力
ファイルからMAX BUF文字読み出す
ファイルの最後か？
まだファイルの途中
ファイルの最後を検出した
ファイルのクローズ
標準エラー出力へエラーメッセージを出力
正常終了時の終了コード設定
エラー終了時の終了コード設定
プログラムの終了処理
プログラム終了

図 4.22　STDFREAD プログラムの処理の流れ

```
;; STDFREAD.ASM File Read Program
;;                Standard I/O version
;;
;;      3DH     Open a File
;;      3EH     Close a File Handle
;;      3FH     Read From File/Device
;;      40H     Write to File/Device
;;      4CH     Terminate a Process

MAXBUF  EQU     1024

CSEG    SEGMENT ............論理セグメントCSEGの始まり
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG ..........セグメントレジスタはCSEGにセットされているとする
        ORG     80H         ⎫ CSEGのオフセット0080Hからにはコマンドラインの
CMDLEN  DB      ?           ⎬ コマンド名を除いたものがはいっている
CMDBUF  DB      127 DUP (?) ⎭ 例）A>STDFREAD FREAD.ASM ⏎ なら…

                  | 10 | ' ' | 'F' | 'R' | 'E' | 'A' | 'D' | '.' | 'A' | 'S' | 'M' | CR |   00
                   文字数 └────────────── 10文字 ──────────────┘

        ORG     100H ..............COM形式の場合は0100H
START:  MOV     BL,CMDLEN   ⎫
        CMP     BL,2        ⎬ コマンドラインの文字数が0、つまり
        JB      NERROR      ⎭ パラメータがなければエラー
        XOR     BH,BH       ⎫ ファイル名（パス名）の末尾の次に00Hを
        MOV     CMDBUF[BX],0⎭ 書き込み、ASCIZ文字列にする
        MOV     DX,OFFSET CMDBUF[1] ..............ファイルオープンのファンクションリクエスト
                                                ⎧AH=3DH
        MOV     AX,3D00H                        ⎨DS：DX=パス名のアドレス
        INT     21H                             ⎩AL=アクセスモード
        JC      NERROR ..............キャリーフラグが「1」ならエラー（オープンできない）
        MOV     BX,AX ..............AXレジスタにファイルハンドルが返される。BXレジスタに保存し、
                                                  以下のループで値を保持する
        MOV     DI,1 ..............標準出力のハンドルを用意する

LOP:    MOV     AH,3FH
        MOV     DX,OFFSET BUF ..............ファイル読み出しのファンクションリクエスト
        MOV     CX,MAXBUF                 ⎧AH=3FH
        INT     21H                       ⎨DS：DX=バッファアドレス
                                          ⎩CX=読み出すバイト数
        TEST    AX,AX   ⎫
        JZ      RDEND   ⎭ 読めなかったら終了
        XCHG    BX,DI ..............標準出力のハンドルをセットする
        MOV     CX,AX ..............読み込んだバイト数をセットする
        MOV     AH,40H ..............ファイル（ハンドル）への書き込みのファンクションリクエスト
        INT     21H                                     ⎧AH=40H
        XCHG    BX,DI ..............ファイルのハンドルをセットする ⎨DS：DX=出力バッファの先頭アドレス
        MOV     SI,CX                                   ⎨CX=書き込むバイト数
                                                        ⎩BX=ファイルハンドル
        CMP     BUF[SI-1],1AH ⎫
        JE      RDEND         ⎭ Ctrl-Z（1AH）で終わっていないか調べる
        JMP     SHORT LOP

RDEND:  MOV     AH,3EH ..............ファイルクローズのファンクションリクエスト
        INT     21H           ⎧AH=3EH
                              ⎩BX=ファイルハンドル
        XOR     AL,AL ..............正常終了の終了コードを設定する
        JMP     SHORT RETURN
```

— リスト 4.6 —（次ページに続く）

```
NERROR: MOV     BX,2
        MOV     CX,OFFSET NERREND - OFFSET NERRMES ……標準エラー出力へ
        MOV     DX,OFFSET NERRMES                    エラーメッセージを出力する
        MOV     AH,40H
        INT     21H
        MOV     AL,1 ……エラー終了の終了コードを設定する
RETURN: MOV     AH,4CH ……プロセス終了のファンクションリクエスト
        INT     21H          {AH=4CH
                             {終了コード

BUF     DB      MAXBUF DUP (?)
NERRMES DB      0DH,0AH,"Not found",0DH,0AH
NERREND EQU     $
CSEG    ENDS
        END     START
```

リスト 4.6　ファンクション 40H のサンプルプログラム STDFREAD（標準出力を利用した FREAD プログラム）

標準出力、標準エラー出力を使ったディスプレイ表示は、ファイルのオープン、クローズの操作がないだけで、FWRITE プログラムで示したファイルへの書き込みとまったく同じ処理内容であることに注目してください（図 4.23）。

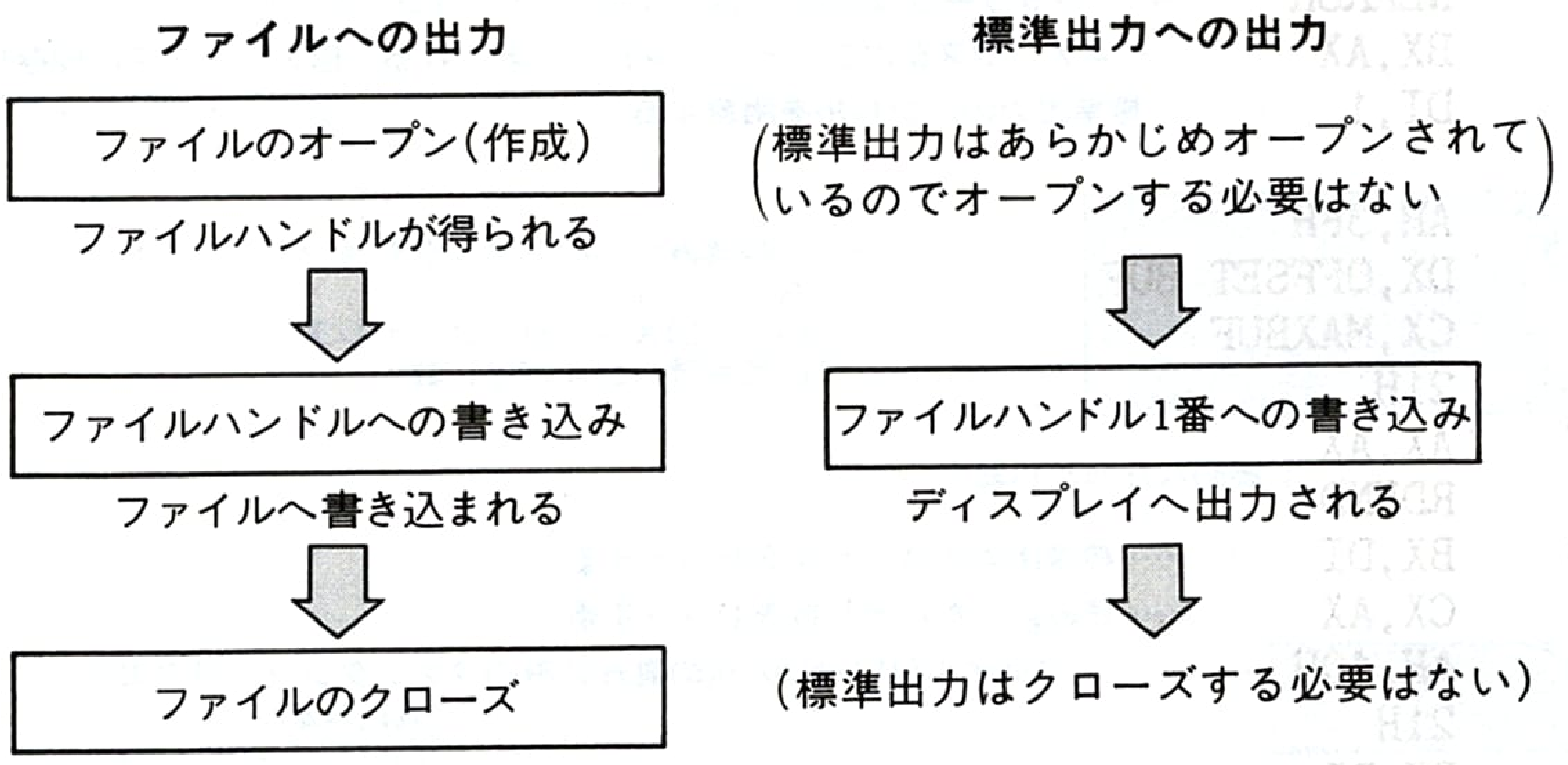
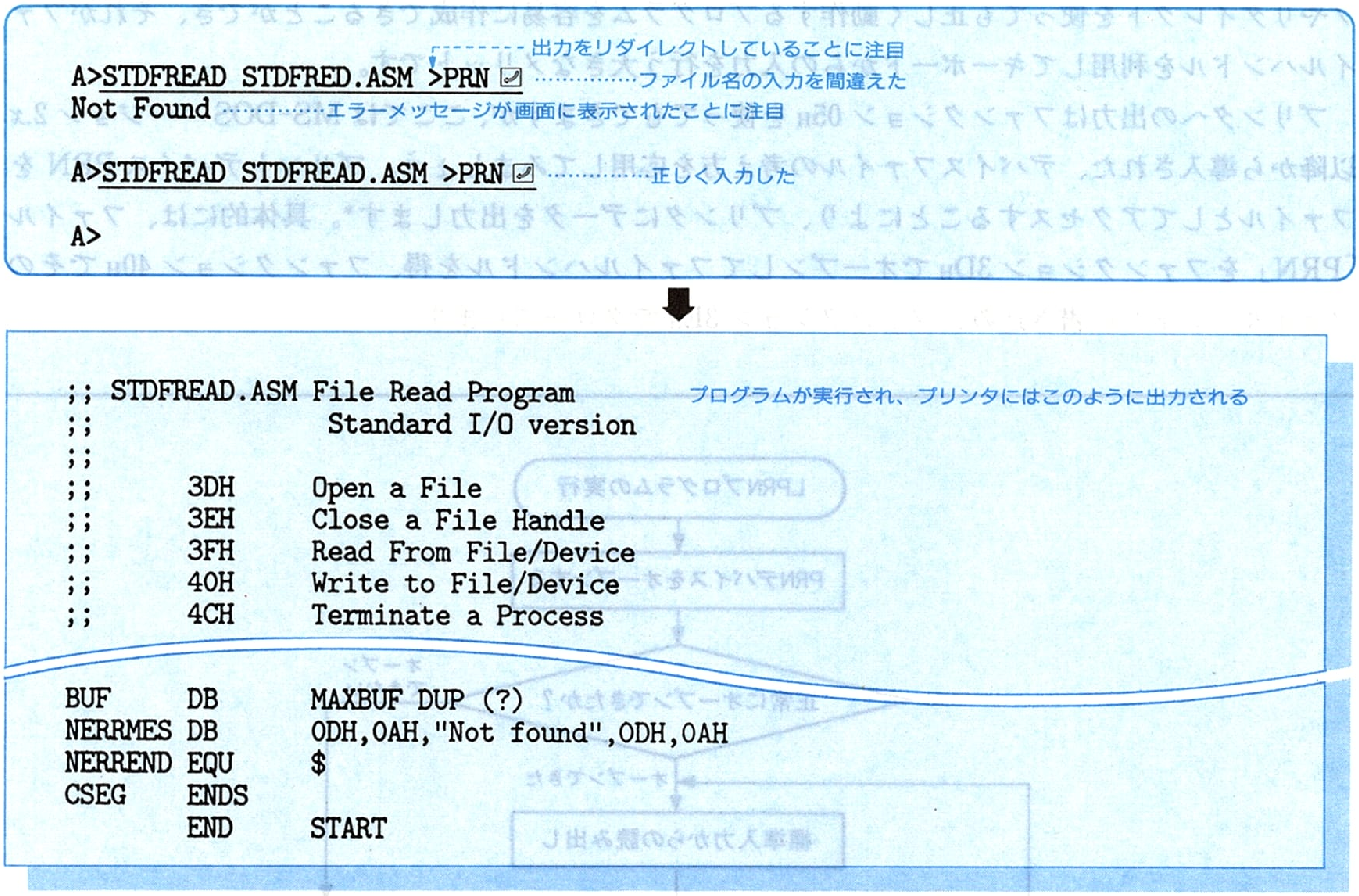

ファイルへの出力
ファイルのオープン（作成）
ファイルハンドルが得られる
ファイルハンドルへの書き込み
ファイルへ書き込まれる
ファイルのクローズ
標準出力への出力
（標準出力はあらかじめオープンされているのでオープンする必要はない）
ファイルハンドル1番への書き込み
ディスプレイへ出力される
（標準出力はクローズする必要はない）

図 4.23　標準出力とファイルへの出力

STDFREAD プログラムの実行例を図 4.24 に示します。誤ったファイル名を指定した場合のエラーメッセージは、標準エラー出力に出力されているために、出力をリダイレクトしてもディスプレイに出力されていることに注目してください。



```
A>STDFREAD STDFRED.ASM >PRN
Not Found

A>STDFREAD STDFREAD.ASM >PRN

A>
```

出力をリダイレクトしていることに注目

ファイル名の入力を間違えた

エラーメッセージが画面に表示されたことに注目

正しく入力した

```
;; STDFREAD.ASM File Read Program
;;                Standard I/O version
;;
;;        3DH     Open a File
;;        3EH     Close a File Handle
;;        3FH     Read From File/Device
;;        40H     Write to File/Device
;;        4CH     Terminate a Process
```

```
BUF      DB      MAXBUF DUP (?)
NERRMES  DB      0DH,0AH,"Not found",0DH,0AH
NERREND  EQU     $
CSEG     ENDS
         END     START
```

プログラムが実行され、プリンタにはこのように出力される

図 4.24　STDFREAD プログラムの実行例

■ ファンクション 3DH、3EH、3FH、40H を利用したプログラム（キーボード入力とプリンタへの出力プログラム LPRN）

［使用ファンクション］

- ファンクション 3DH ……… ファイルのオープン
- ファンクション 3EH ……… ファイルのクローズ
- ファンクション 3FH ……… ハンドルの読み出し
- ファンクション 40H ……… ハンドルの書き込み

キーボードからの入力データの読み出しも、標準入力という形でファイルと同じような方法で入力することができます。ファイルハンドルの 0 番は標準入力としてあらかじめオープンされているので、ファンクション 3FH を使って読み出せばよいのです。

この方法を使うと、行末のリターンキーの入力は 0DH、0AH という 2 バイトの文字に変換されます。これは MS-DOS のテキストファイルの形式と同じであるため、リダイレクトにより入力をファイルから行っても同じ形式でデータが得られることになります。キーボードからの入力だけでなく、パイ

プやリダイレクトを使っても正しく動作するプログラムを容易に作成できることができ、それがファイルハンドルを利用してキーボードからの入力を行う大きなメリットです。

プリンタへの出力はファンクション 05Hを使ってもできますが、ここでは MS-DOS バージョン 2.x 以降から導入された、デバイスファイルの考え方を応用してみましょう。プリントデバイス PRN をファイルとしてアクセスすることにより、プリンタにデータを出力します*。具体的には、ファイル「PRN」をファンクション 3DHでオープンしてファイルハンドルを得、ファンクション 40Hでそのファイルハンドルに書き込み、ファンクション 3EHでクローズします。

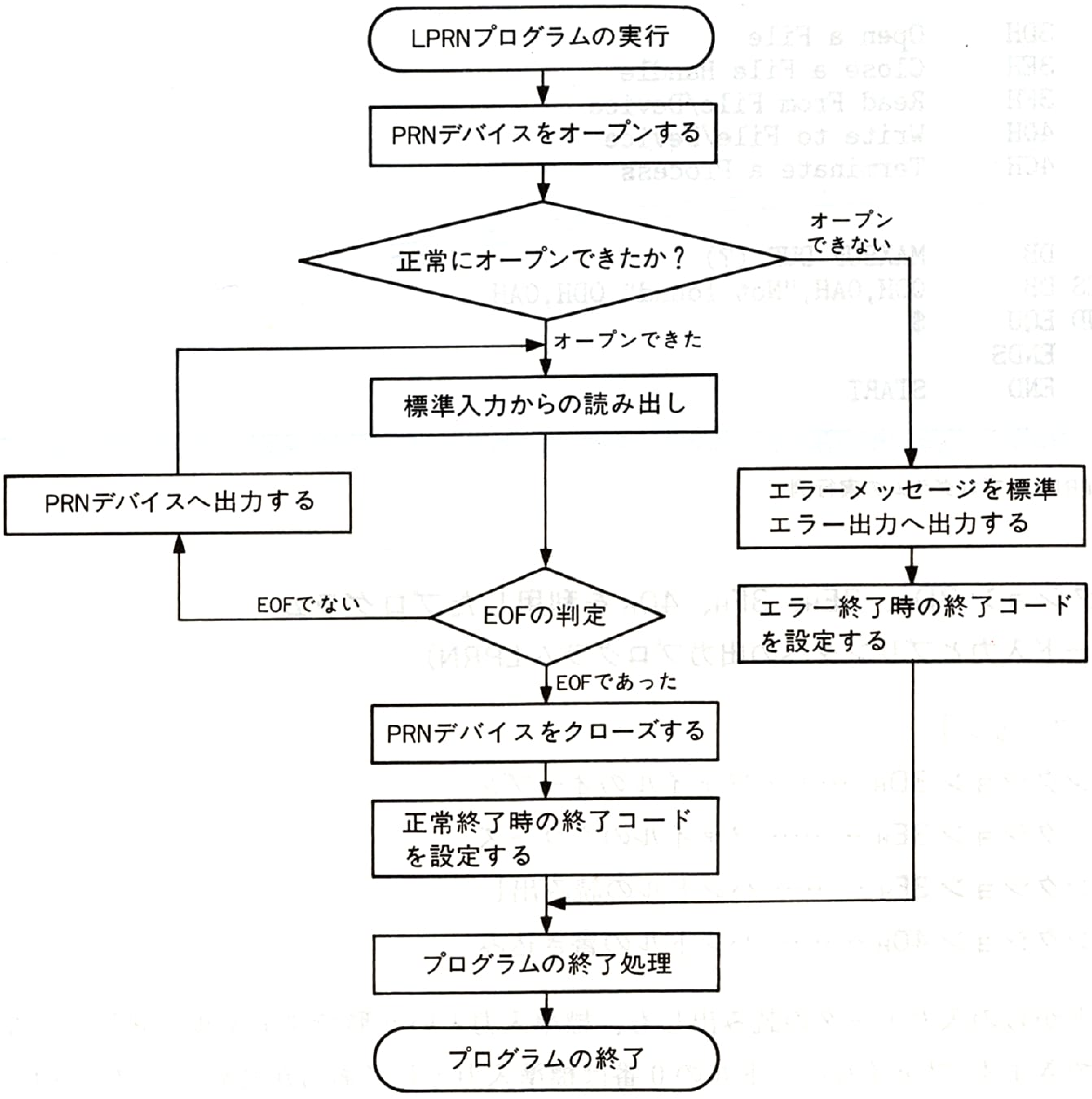

LPRNプログラムの実行
PRNデバイスをオープンする
正常にオープンできたか？
オープンできない
オープンできた
標準入力からの読み出し
PRNデバイスへ出力する
EOFでない
EOFの判定
EOFであった
PRNデバイスをクローズする
正常終了時の終了コードを設定する
エラーメッセージを標準エラー出力へ出力する
エラー終了時の終了コードを設定する
プログラムの終了処理
プログラムの終了

図 4.25　LPRN プログラムの処理の流れ

---

* バージョン 3.1 以降の PC-9801 シリーズ用 MS-DOS には、PRINT.SYS を CONFIG.SYS に組み込んでおかないと PRN デバイスが利用できないものがある。

デバイスをファイルとしてアクセスするという考え方はUNIXから取り入れられたものです。このことを実現したプログラム「LPRN」のソースファイルを**リスト4.7**に、その処理の流れを**図4.25**に、実行例を**図4.26**に示します。デバイスをファイルとして扱うことによって、さまざまなデバイスを統一的に扱うことができ、１つのプログラムをファイルにもデバイスにも利用しやすくなります。

```
;; LPRN.ASM     Line Input and Print Out
;;
;;      3DH     Open a File
;;      3EH     Close a File Handle
;;      3FH     Read From File/Device
;;      40H     Write to File/Device
;;      4CH     Terminate a Process

CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG

        ORG     100H
START:  MOV     DX,OFFSET PRN
        MOV     AX,3D01H          PRNデバイスを書き込みモードでオープンする
        INT     21H
        JC      NERROR
        MOV     DI,AX       ……………ハンドルを退避する
        XOR     BX,BX       ……………標準入力のハンドルをセットする
        MOV     DX,OFFSET BUF

LOP:    MOV     AH,3FH
        MOV     CX,1          ……………標準入力から1バイト読み出す
        INT     21H
        XCHG    BX,DI       ……………PRNデバイスのハンドルをセットする
        TEST    AX,AX
        JE      RDEND
        CMP     BUF,1AH       EOFならば終了処理へ
        JE      RDEND
        MOV     CX,AX
        MOV     AH,40H        ……………PRNデバイスに出力する
        INT     21H
        XCHG    BX,DI       ……………標準入力のハンドルをセットする
        JMP     SHORT LOP

RDEND:  MOV     AH,3EH        ……………PRNデバイスをクローズする
        INT     21H
        XOR     AL,AL           正常終了の終了コードを設定する
        JMP     SHORT RETURN

NERROR: MOV     BX,2
        MOV     DX,OFFSET NERRMES
        MOV     CX,OFFSET NERREND - OFFSET NERRMES  ……標準エラー出力にエラーメッセージ
        MOV     AH,40H                                  を出力する
        INT     21H
```

— リスト 4.7 —（次ページに続く）

```
          MOV     AL,1  ……………エラー終了の終了コードを設定する
 RETURN:  MOV     AH,4CH
          INT     21H   ……………プログラムを終了する

BUF       DB      0
NERRMES   DB      0DH,0AH,"PRN: Can't open",0DH,0AH
NERREND   EQU     $
PRN       DB      "PRN",0
CSEG      ENDS
          END     START
```

リスト 4.7　ファンクション $3F_{H}$、40_{H} のサンプルプログラム LPRN

```
A>LPRN ⏎  ……………LPRNコマンドを実行する
These are characters printed on paper.⏎  ……………キーボードから、任意の文字を入力する
                                                    ⏎の入力により印字が開始される
^Z ⏎  ……………Ctrl-Z⏎でコマンドラインに戻る

A>
```

⬇

このように印字される

```
These are characters printed on paper.
```

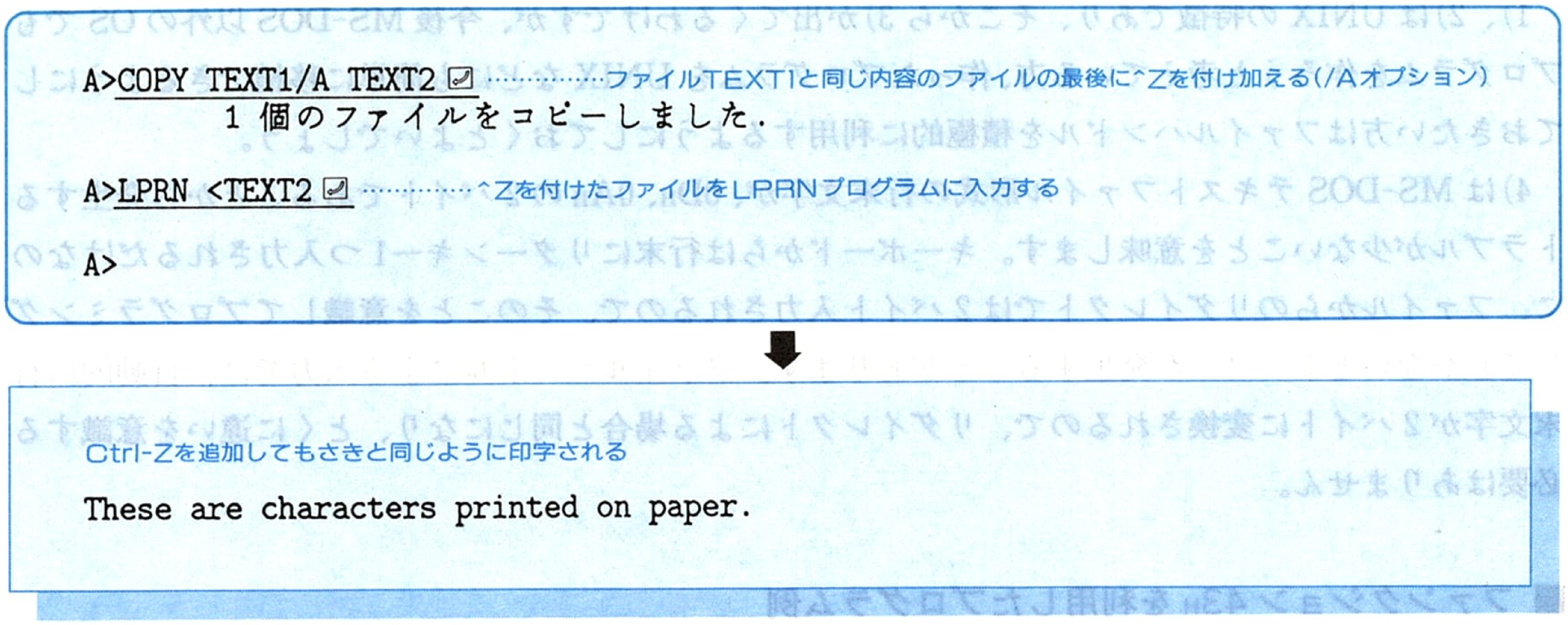
```
A>COPY CON TEXT1 ⏎  ……………実験用のファイルTEXT1を作成する（この場合、ファイルの最後に^Zは付かない）
These are characters printed on paper.⏎
^Z⏎
        1 個のファイルをコピーしました.

A>LPRN <TEXT1 ⏎  ……………リダイレクトによりファイルTEXT1をLPRNプログラムに入力する

A>
```

⬇

このように印字される

```
These are characters printed on paper.
```

— 図 4.26 —（次ページに続く）



```
A>COPY TEXT1/A TEXT2⏎
      1 個のファイルをコピーしました.

A>LPRN <TEXT2⏎

A>
```

……ファイルTEXT1と同じ内容のファイルの最後に^Zを付け加える(/Aオプション)

……^Zを付けたファイルをLPRNプログラムに入力する

Ctrl-Zを追加してもさきと同じように印字される

```
These are characters printed on paper.
```

図 4.26　LPRN プログラムの実行例

実際に LPRN プログラムを実行してみるとわかりますが、キーボードからの入力中はコマンドラインでの入力と同じようにバックスペースキーによる編集や、ファンクションキーを用いて1つ前の行を呼び出すテンプレート編集が利用できます。これはコンソールドライバ(CON デバイスのデバイスドライバ)が行内編集機能を提供しているからです。

コンソールドライバは通常、1行単位で編集された結果の文字列(COOKED モード)を返しますが、押されたキーの文字コードをすべて引き渡す(RAW モード)ことも可能です。本書では解説しませんが、ファンクション 44_H の IOCTL(I／O コントロール)機能を利用すれば、入力モードを切り換えたり文字が入力されているかどうかを調べたりすることができます*。

ファイルハンドルを使った入出力と、この IOCTL を組み合わせることによって、ファンクション 01_H 番から $0C_H$ 番までの機能をすべて実現することができます。バージョン 1.25 時代の古いファンクションは使う必要がなく、すべてバージョン 2.*x* からの新しいファンクションで置き換えることができるのです。というより、できるだけ新しいファンクションを利用すべきでしょう。

プログラムの入出力をファイルハンドルによる入出力に統一することには次のようなメリットがあります。

1) デバイスもファイルも同じ処理手順でプログラミングができる
2) 同じルーチンをファイルにもデバイスにも利用することができる
3) UNIX や OS/2 などの OS でも同じ考え方でプログラミングができる
4) リダイレクトとの相性がよい

---

* RAW(ロー)モードの利用法や IOCTL については、『MS-DOS プログラミングテクニック』や『応用 C 言語』(ともにアスキー出版局発行)などを参考にするとよいでしょう。

1)、2)はUNIXの特徴であり、そこから3)が出てくるわけですが、今後MS-DOS以外のOSでもプログラムを作ろうと考えている方、作ったプログラムをUNIXなどにも簡単に移植できるようにしておきたい方はファイルハンドルを積極的に利用するようにしておくとよいでしょう。

4)はMS-DOSテキストファイル形式の行末文字が、0D$_\mathrm{H}$、0A$_\mathrm{H}$の2バイトであることから発生するトラブルが少ないことを意味します。キーボードからは行末にリターンキー1つ入力されるだけなのに、ファイルからのリダイレクトでは2バイト入力されるので、そのことを意識してプログラミングしておかないとトラブルが発生することがあります。ファイルハンドルによる入力では、自動的に行末文字が2バイトに変換されるので、リダイレクトによる場合と同じになり、とくに違いを意識する必要はありません。

■ ファンクション43$_\mathrm{H}$を利用したプログラム例
（1章のファイル属性設定プログラムCHMOD）

［使用ファンクション］

ファンクション43$_\mathrm{H}$ ……… ファイル属性の取り出し/設定

1章で作成したファイル属性設定プログラムCHMODプログラムの全体について、ファンクションリクエストを中心に、ソースファイル上で詳しく解説しましょう（**リスト4.8**）。

```
;; CHMOD.ASM      CHANGE FILE ATTRIBUTES
;;
;;        09H     Display String
;;        43H     Change Attributes
;;        4CH     Terminate a Process

PRINT     MACRO   MSGADR
          MOV     AH,09H
          MOV     DX,OFFSET MSGADR
          INT     21H
          ENDM

CSEG      SEGMENT
          ASSUME  CS:CSEG,DS:CSEG,ES:CSEG

          ORG     80H
CMDLEN    DB      ?
CMDBUF    DB      127 DUP (?)

          ORG     100H
```

（MOV AH,09H ～ INT 21H）……ディスプレイへの文字列出力のファンクションリクエスト

（PRINT MACRO MSGADR ～ ENDM）文字列を出力するというマクロ定義(5章参照)　PRINT　メッセージアドレス　と記述することによりこの3行が展開される

（CSEG SEGMENT）……論理セグメントCSEGの始まり

（ASSUME CS:CSEG,DS:CSEG,ES:CSEG）……セグメントレジスタはCSEGにセットされているとする

（ORG 80H ～ CMDBUF DB 127 DUP (?)）0080$_\mathrm{H}$からのコマンドラインの文字列から、コマンド名CHMODを除いたものがセットされる

（ORG 100H）……COM形式は必ず0100$_\mathrm{H}$

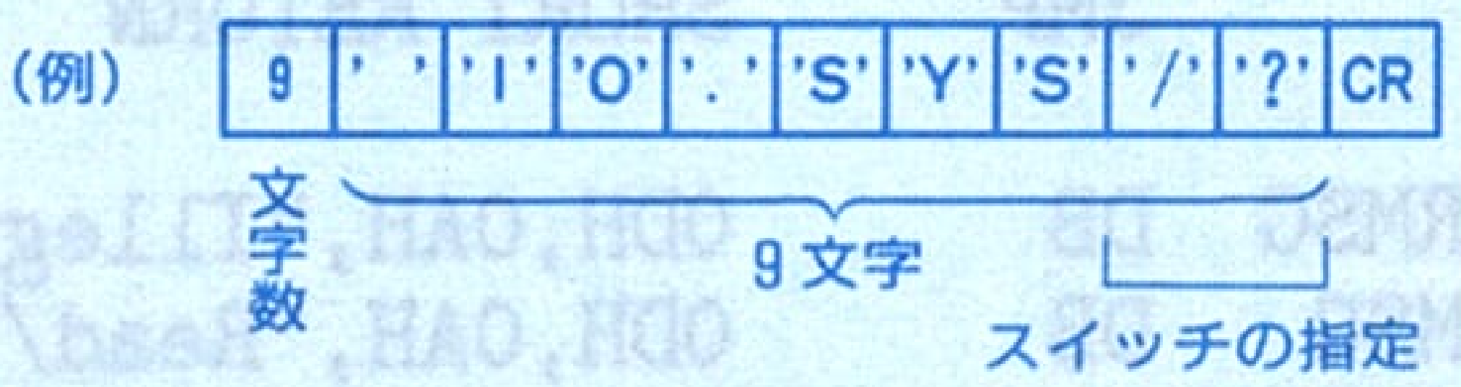
— リスト4.8 —（次ページ以下に続く）

```
START:
        MOV     BL,CMDLEN          コマンドラインのパラメータの先頭に入力文字数が
        CMP     BL,4               はいっている。先頭は空白であり、またスイッチに
        JB      ERROR              2文字必要とするので、入力されたパラメータは3
                                   文字より多くなければ入力エラーになる
        XOR     BH,BH
        CMP     CMDBUF[BX-2],'/'   コマンドラインのパラメータの後ろから2つ目が
        JNE     ERROR              「/」でなければスイッチ指定の誤り
        MOV     DX,OFFSET CMDBUF[1]
        MOV     CMDBUF[BX-2],0     ……ファイル属性設定のファンクションリクエスト
                                        AH=43H
                                        AL=00H(取得)、01H(設定)
        MOV     AX,4300H                DS:DX=パス名のアドレス
        INT     21H                     CX=属性
        JC      ERROR              ……キャリーフラグが「1」ならエラー(ファイルが存在しない)
        TEST    CX,00011000B       ディレクトリ、ボリュームラベルで
        JNE     ERROR              あればエラー(ファイルでない)
GETSW:
        MOV     AL,CMDBUF[BX-1]    ……スイッチの文字を得る
        CMP     AL,'?'
        JE      ?SWITCH
        AND     AL,NOT('A' XOR 'a')   ……小文字を大文字に変換する
        CMP     AL,'R'                  (大文字ならばそのまま)
        JE      RSWITCH
        CMP     AL,'W'
        JE      WSWITCH
        CMP     AL,'H'             各スイッチに関する処理へ分岐
        JE      HSWITCH
        CMP     AL,'N'
        JE      NSWITCH
        JMP     SHORT ERROR        ……定義されていないスイッチを指定した

RSWITCH:
        OR      CX,00000001B       ……リードオンリーのビットを「1」にする
        JMP     SHORT CHANGE
WSWITCH:
        AND     CX,11111110B       ……リードオンリーのビットを「0」にする
        JMP     SHORT CHANGE         (リードライト可能にする)
HSWITCH:
        OR      CX,00000010B       ……隠しファイル属性のビットを「1」にする
        JMP     SHORT CHANGE         (隠しファイルにする)
NSWITCH:
        AND     CX,11111101B       ……隠しファイル属性のビットを「0」にする
CHANGE:                              (見えるファイルにする)
        MOV     AX,4301H           ……ファイル属性変更のファンクション
        INT     21H                  リクエスト
        JC      ERROR                 AH=43H
        XOR     AL,AL                 AL=00H(取得)、01H(設定)
        JMP     SHORT RETURN          DS:DX=パス名のアドレス
                                      CX=属性
                                   キャリーフラグが「1」ならエラー(実際には
                                   ここでエラーになることはないはず)

ERROR:
        PRINT   ERRMSG             ……エラーメッセージの出力
        MOV     AL,1
```

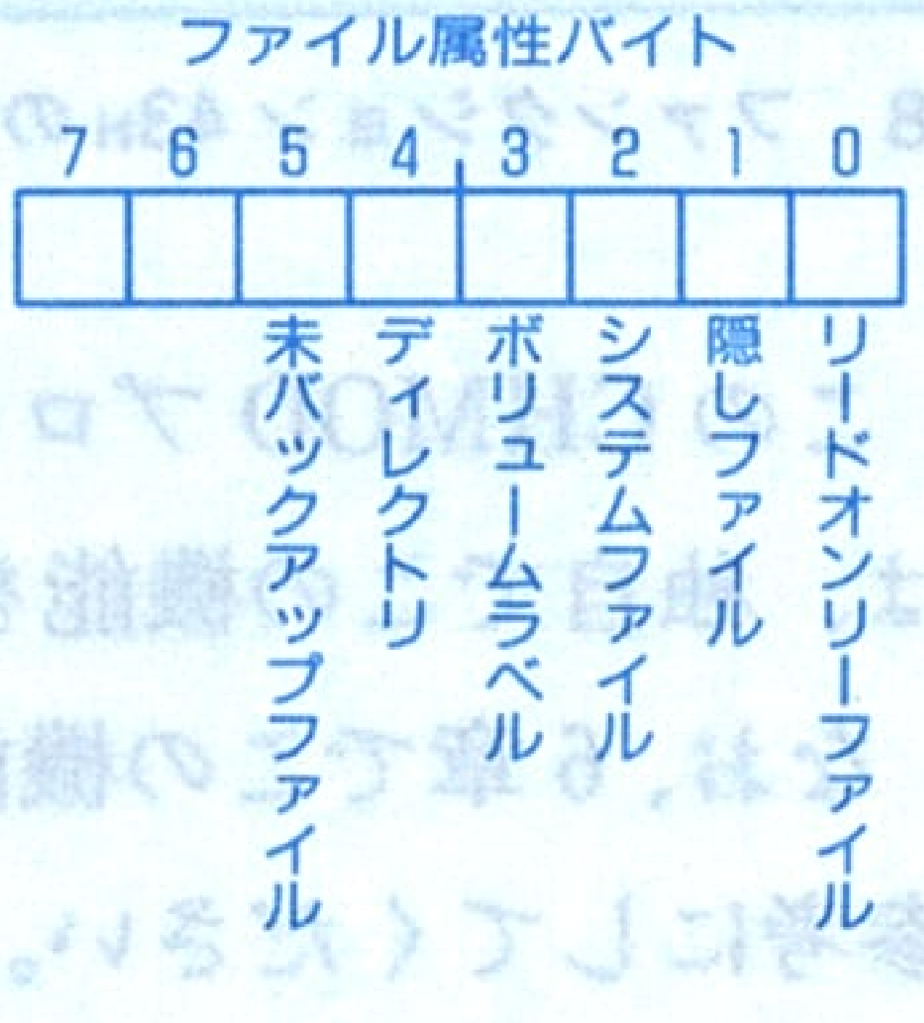
ファイルの属性を得る

初期設定

スイッチの解析



/R、/W、/H、/Nスイッチの処理、各スイッチに応じて属性のビットを操作する



それぞれ「1」が立ったビットの位置の属性が有効となる

変更した属性をファイルにセットする

エラー処理

```
RETURN:
          MOV       AH,4CH
          INT       21H ……………プロセス終了のファンクションリクエスト
                                   {AH=4CH
                                   {AL=終了コード
?SWITCH:
          TEST      CX,00000001B } リードオンリーの属性のビットが「1」か否か
          JE        RWFILE       }
          PRINT     ROMSG ……………「Read Only」と表示する
          JMP       SHORT HDCHK
RWFILE:
          PRINT     RWMSG ……………「Read/Write」と表示する
HDCHK:
          TEST      CX,00000010B } 隠しファイルの属性のビットが「1」か否か
          JE        SYSCHK       }
          PRINT     HDMSG ……………「Hidden」と表示する
SYSCHK:
          TEST      CX,00000100B } システムファイルの属性ビットが「1」か否か
          JE        CHKEND       }
          PRINT     SYSMSG ……………「System」と表示する
CHKEND:
          PRINT     FILEMSG ……………最後に「File」と表示する
          XOR       AL,AL
          JMP       SHORT RETURN

ERRMSG    DB        0DH,0AH,"Illegal switch or filename",0DH,0AH,'$'
RWMSG     DB        0DH,0AH,"Read/Write ",'$' ……………リード/ライトのメッセージ
ROMSG     DB        0DH,0AH,"Read Only ",'$' ……………リードオンリーのメッセージ
HDMSG     DB        "Hidden ",'$' ……………hiddenのメッセージ
SYSMSG    DB        "System ",'$' ……………システムのメッセージ
FILEMSG   DB        "File",0DH,0AH,'$' ……………ファイルというメッセージ

CSEG      ENDS ……………論理セグメントCSEGの終わり
          END       START ……………プログラム開始アドレスを指定する
```

/?スイッチの処理属性の各ビットを調べて対応するメッセージを出力する

復帰改行コード CR/LF

文字列の最後は「$」で終了する

各種のメッセージ

リスト 4.8　ファンクション 43H のサンプルプログラム（1 章の CHMOD プログラム）

なお、この CHMOD プログラムにはシステム属性についてはサポートされていませんが、すでにみなさんは、独自でこの機能を追加できる知識が備わっていると思いますので、各自で挑戦してみてください。なお、6 章でこの機能をサポートした CHMOD プログラムの拡張版を C 言語で作成していますので参考にしてください。

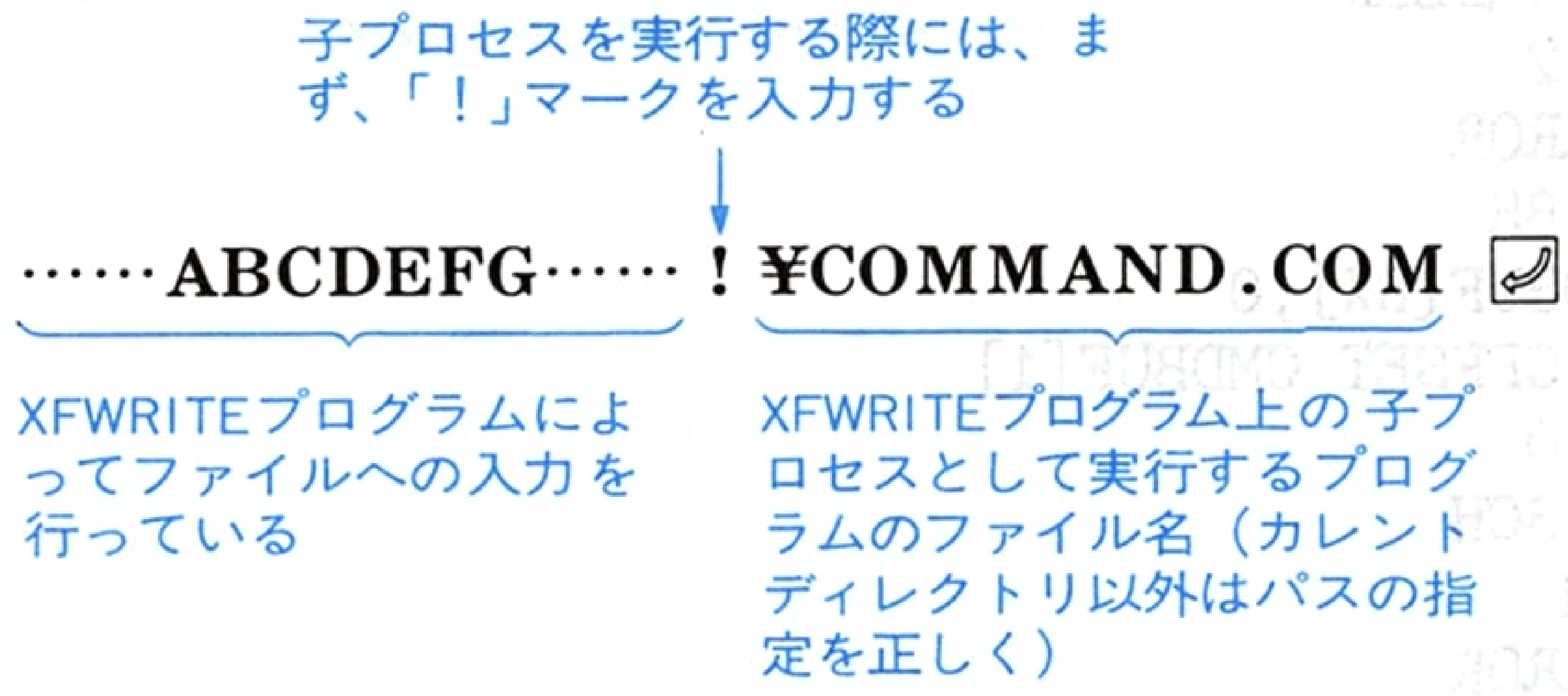
■ ファンクション 4AH、4BH を利用したプログラム例（子プロセスの起動プログラム）

［使用ファンクション］

ファンクション 4AH ……… メモリブロックサイズの変更
ファンクション 4BH ……… プログラムのロードと実行

このサンプルプログラムは、2章で作成したファイルの新規作成プログラムFWRITEを親プロセスとして、任意の子プロセスを起動できるように、FWRITEプログラムにその機能を追加したものです。このプログラムを「XFWRITE」としましょう。

このXFWRITEプログラムの実行中に子プロセスを起動するには、FWRITEプログラムと同様にキーボードからファイルへ入力しているときに、まず「！」を入力し、それに続けて実行しようとするプログラムのファイル名(カレントディレクトリ以外はパスを含む)を入力します。この操作により、XFWRITEプログラム上から任意のプログラムが実行できます。たとえば、次のようにすると、ファイルのキー入力の途中でCOMMAND.COMが実行されますので、DIRコマンドなどの内部コマンドを実行することができます。起動したプログラムの実行が終われば、もとのキー入力に戻ります。



子プロセスを実行する際には、まず、「！」マークを入力する

XFWRITEプログラムによってファイルへの入力を行っている

XFWRITEプログラム上の子プロセスとして実行するプログラムのファイル名（カレントディレクトリ以外はパスの指定を正しく）

では、**リスト4.9**に示すXFWRITEプログラムのソースファイルをもとに、プロセス管理、メモリ管理に関するファンクションリクエストの使われ方について解説しましょう。

```
;; XFWRITE.ASM  File Write Program
;;               With Exec Process
;;
;;      01H     Read Keyboard and Echo
;;      02H     Display Character
;;      09H     Display String
;;      0AH     Buffered Keyboard Input
;;      3CH     Create a File
;;      3EH     Close a File Handle
;;      40H     Write to File/Device
;;      4AH     Modify Allocated Memory
;;      4BH     Load/Execute a Program
;;      4CH     Terminate a Process

CSEG    SEGMENT
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG
STKSIZE EQU     256
        ORG     80H
```

— リスト4.9 —（次ページ以下に続く）

```
CMDLEN  DB      ?
CMDBUF  DB      127 DUP (?)

        ORG     100H
START:  MOV     SP,OFFSET CODEEND+STKSIZE
        MOV     BX,OFFSET CODEEND+STKSIZE
        MOV     CL,4
        SHR     BX,CL
        INC     BX
        MOV     AH,4AH
        INT     21H
        MOV     BLOCK[4],DS
        MOV     BLOCK[8],DS
        MOV     BLOCK[12],DS

        MOV     BL,CMDLEN
        CMP     BL,2
        JB      CERROR
        XOR     BH,BH
        MOV     CMDBUF[BX],0
        MOV     DX,OFFSET CMDBUF[1]
        MOV     CX,0
        MOV     AH,3CH
        INT     21H
        JC      CERROR
        MOV     BX,AX

LOP:    MOV     AH,01H
        INT     21H
        CMP     AL,'!'
        JE      CHILD
        MOV     BUF,AL
        MOV     DX,OFFSET BUF
        MOV     CX,1
        MOV     AH,40H
        INT     21H
        CMP     BUF,1AH
        JE      WRTEND
        CMP     BUF,0DH
        JNE     LOP
        MOV     DL,0AH
        MOV     AH,02H
        INT     21H
        MOV     BUF,0AH
        MOV     DX,OFFSET BUF
        MOV     AH,40H
        MOV     CX,1
        INT     21H
        JMP     SHORT LOP
```

▨の部分がFWRITE.ASMを変更、追加した箇所、
その他はFWRITEのソースファイル参照

スタック領域をコード領域の直後に設定する

メモリブロックの再割り当てのファンクションリクエスト
AH＝$4A_H$
ES＝メモリブロックの先頭
BX＝割り当てを要求するブロックのパラグラフ数

コード＋スタック以外のメモリを解放する（COMモデルではすべてのメモリがプロセスに割り当てられているため）

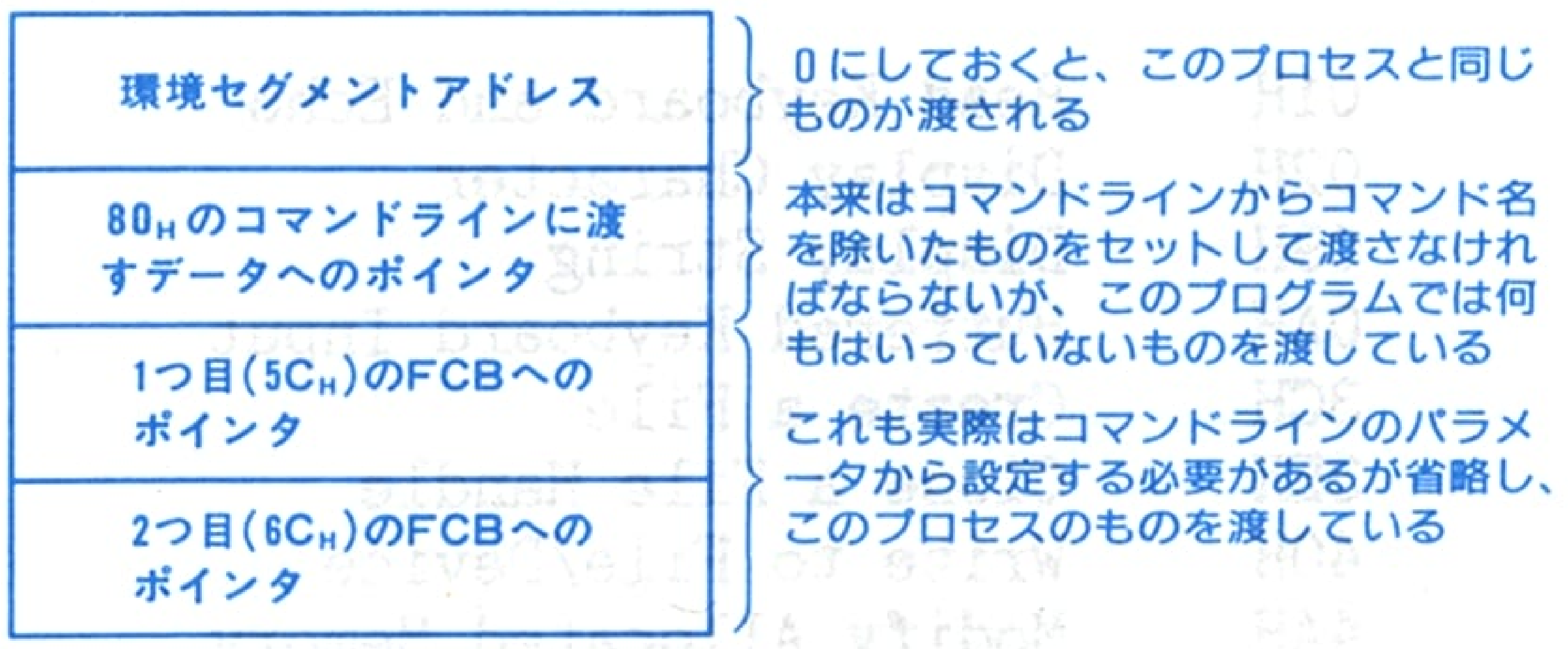
（コード＋スタック）/16＋1がパラグラフサイズ

プロセス自身のメモリブロックの先頭はPSPであるから、そのセグメントアドレスはすでにESレジスタにセットされている

子プロセスを呼び出すときに必要なパラメータブロックのセグメントアドレスをセットしておく

「！」が入力されると子プロセスの呼び出し処理へ移る

パラメータブロック


環境セグメントアドレス
80_Hのコマンドラインに渡すデータへのポインタ
1つ目($5C_H$)のFCBへのポインタ
2つ目($6C_H$)のFCBへのポインタ

0にしておくと、このプロセスと同じものが渡される

本来はコマンドラインからコマンド名を除いたものをセットして渡さなければならないが、このプログラムでは何もはいっていないものを渡している

これも実際はコマンドラインのパラメータから設定する必要があるが省略し、このプロセスのものを渡している

パラメータブロックとしては最低限に簡略化したものを渡しているので、起動するプログラムにはかなりの制限がある。COMMAND.COMと同じような機能を持たせるには、コマンド名に「.COM」「.EXE」を付けたものをPATH順に探し、各パラメータをセットする必要がある

```
WRTEND: MOV     AH,3EH
        INT     21H
        XOR     AL,AL
        JMP     SHORT RETURN

CERROR: MOV     DX,OFFSET CERRMES
        MOV     AH,09H
        INT     21H
        MOV     AL,1
RETURN: MOV     AH,4CH
        INT     21H

CHILD:  PUSH    BX
        MOV     DX,OFFSET PROCESS
        MOV     AH,0AH
        INT     21H
        MOV     DL,0AH
        MOV     AH,02H
        INT     21H
        MOV     BL,PROCESS[1]
        XOR     BH,BH
        MOV     PROCESS[2][BX],0
        MOV     DX,OFFSET PROCESS[2]
        MOV     BX,OFFSET BLOCK
        MOV     AX,4B00H
        INT     21H
        JNC     CHRET
        MOV     DX,OFFSET EERRMES
        MOV     AH,09H
        INT     21H
CHRET:  POP     BX
        JMP     LOP

BUF     DB      0
CERRMES DB      0DH,0AH,"Can't create",0DH,0AH,'$'
EERRMES DB      0DH,0AH,"Can't execute",0DH,0AH,'$'
PROCESS DB      128,0,128 DUP (?)
BLOCK   DW      0000H,OFFSET PARAM,?,5CH,?,6CH,?
PARAM   DW      0,0DH,7EH DUP (?)
CODEEND EQU     $

CSEG    ENDS
        END     START
```

子プロセスの呼び出し処理

PUSH BX……ファイルハンドルを格納してあるBXレジスタを保存する

MOV DX,OFFSET PROCESS～INT 21H……バッファードキーボード入力のファンクションリクエスト
{AH=0A$_H$
{DS：DX=入力バッファ
} 実行するプログラムのパス名を入力する

MOV DL,0AH～INT 21H……ディスプレイへ1文字出力するファンクションリクエスト
{AH=02$_H$
{DL=出力する文字
} 上のコールではCRしかエコーされないのでLFを出力する

MOV BL,PROCESS[1]～MOV PROCESS[2][BX],0……入力されたパス名の末尾に00$_H$を加えてASCIZ文字列にする

MOV DX,OFFSET PROCESS[2]～INT 21H……プログラムのロードと実行のファンクションリクエスト
{AH=4B$_H$
{AL=00$_H$(ロードと実行)、01$_H$(ロードのみ)
{DS：DX=パス名のアドレス
{ES：BX=パラメータブロックのアドレス
} 入力された名前のファイル名を持つプログラムをロードし、実行する

POP BX……ファイルハンドルを格納していたBXレジスタを復帰する

JMP LOP……もとの処理に戻る

PROCESS……ロードするプログラム名を格納する

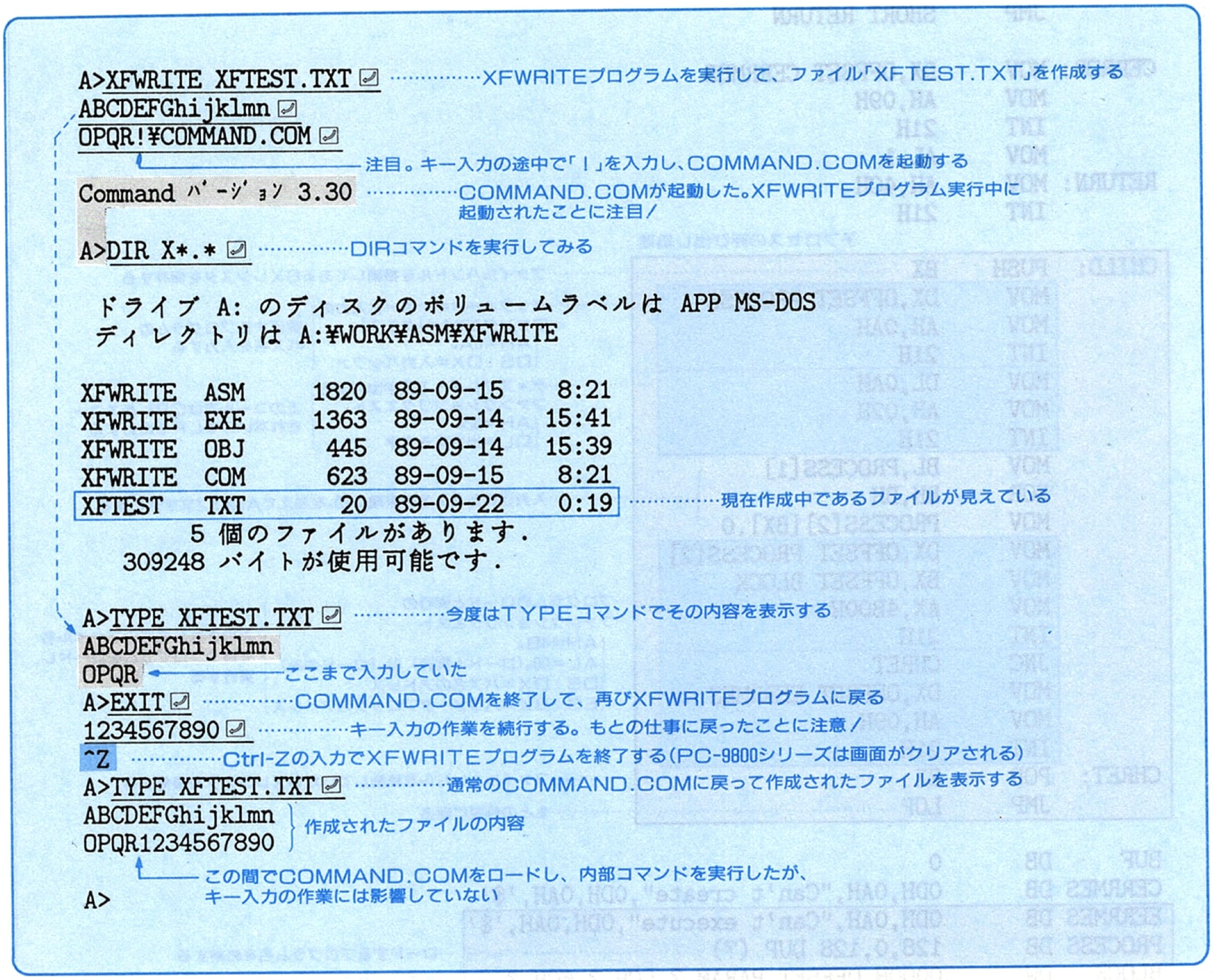
BLOCK……パラメータブロック

PARAM……ロードするプログラムに渡すコマンドラインの内容

CODEEND……コード領域の終わりを示すためのラベル

リスト4.9　ファンクション4AH、4BHのサンプルプログラムXFWRITEのソースファイル
(2章のFWRITEプログラムに機能を追加)

図 4.27 に、完成した XFWRITE プログラム(「XFWRITE.COM」。作成手順は今までのものと同じ)の実行例を示します。

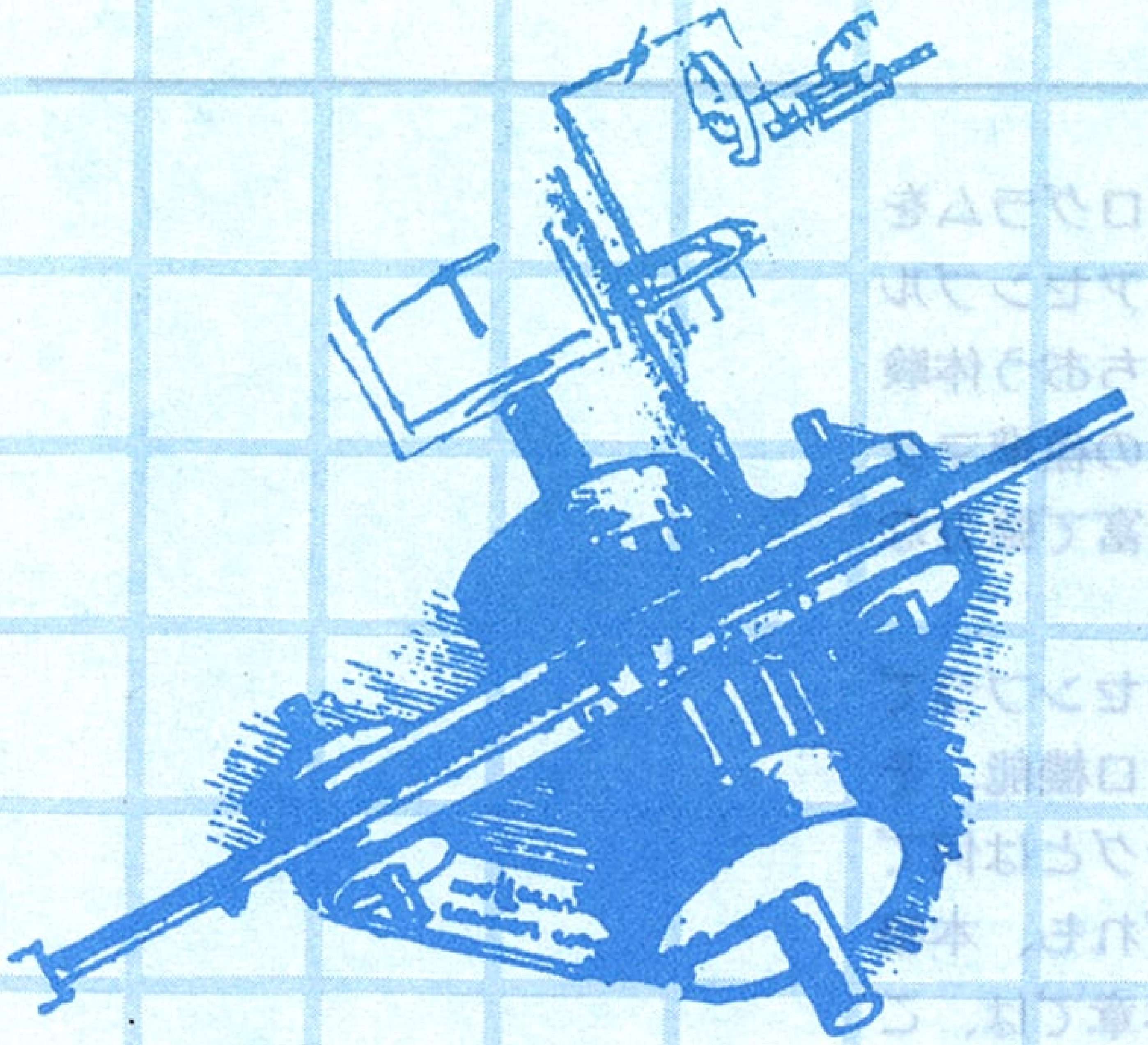


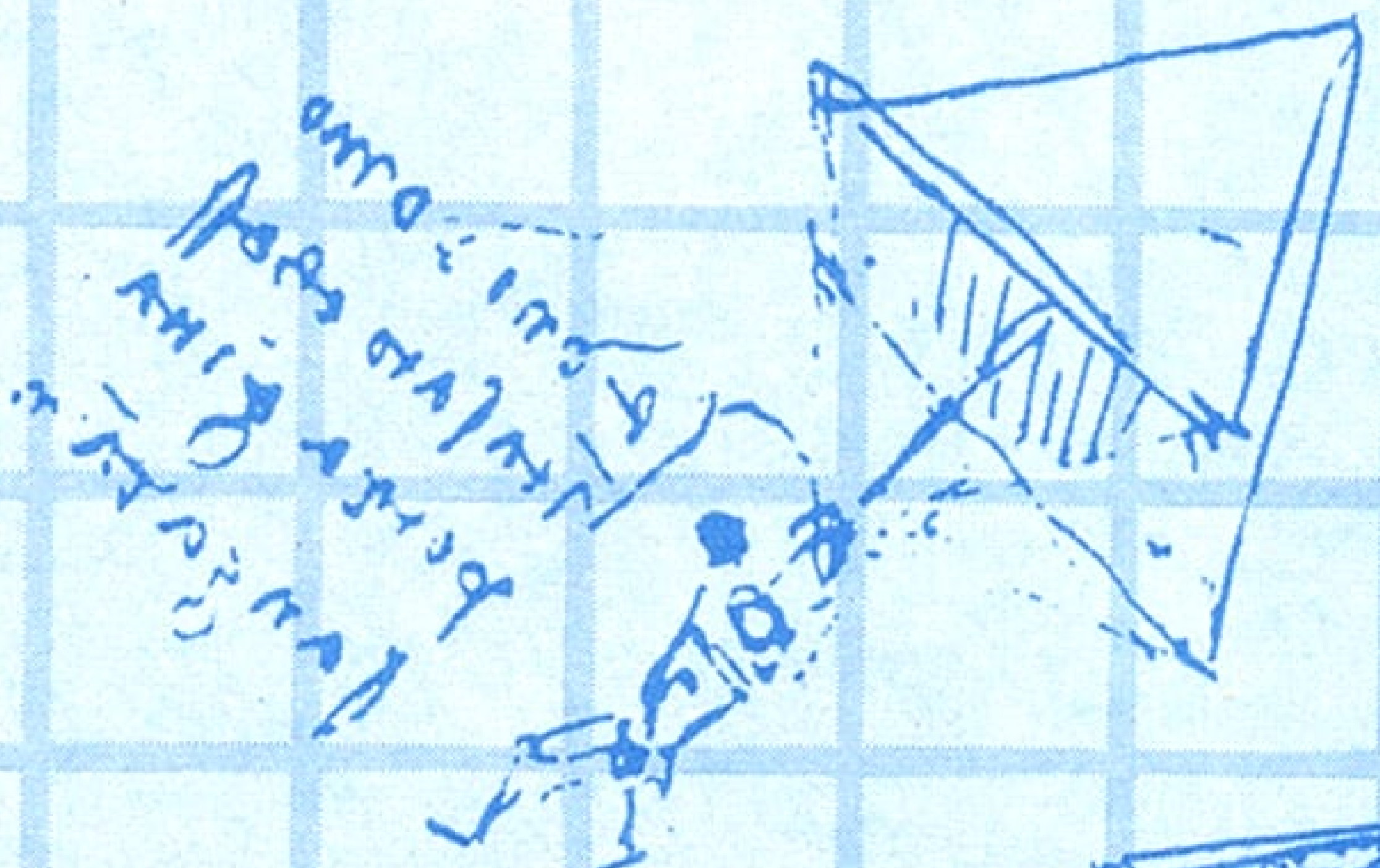
```
A>XFWRITE XFTEST.TXT ↵ ……………XFWRITEプログラムを実行して、ファイル「XFTEST.TXT」を作成する
ABCDEFGhijklmn ↵
OPQR!¥COMMAND.COM ↵
    └── 注目。キー入力の途中で「！」を入力し、COMMAND.COMを起動する
Command ﾊﾞｰｼﾞｮﾝ 3.30 ……………COMMAND.COMが起動した。XFWRITEプログラム実行中に
                                起動されたことに注目／

A>DIR X*.* ↵ ……………DIRコマンドを実行してみる

 ドライブ A: のディスクのボリュームラベルは APP MS-DOS
 ディレクトリは A:¥WORK¥ASM¥XFWRITE

XFWRITE  ASM     1820  89-09-15   8:21
XFWRITE  EXE     1363  89-09-14  15:41
XFWRITE  OBJ      445  89-09-14  15:39
XFWRITE  COM      623  89-09-15   8:21
XFTEST   TXT       20  89-09-22   0:19 ……………現在作成中であるファイルが見えている
        5 個のファイルがあります.
    309248 バイトが使用可能です.

A>TYPE XFTEST.TXT ↵ ……………今度はTYPEコマンドでその内容を表示する
ABCDEFGhijklmn
OPQR ←――――ここまで入力していた
A>EXIT ↵ ……………COMMAND.COMを終了して、再びXFWRITEプログラムに戻る
1234567890 ↵ ……………キー入力の作業を続行する。もとの仕事に戻ったことに注意／
^Z ……………Ctrl-Zの入力でXFWRITEプログラムを終了する(PC-9800シリーズは画面がクリアされる)
A>TYPE XFTEST.TXT ↵ ……………通常のCOMMAND.COMに戻って作成されたファイルを表示する
ABCDEFGhijklmn  }
OPQR1234567890  } 作成されたファイルの内容
    └── この間でCOMMAND.COMをロードし、内部コマンドを実行したが、
        キー入力の作業には影響していない
A>
```

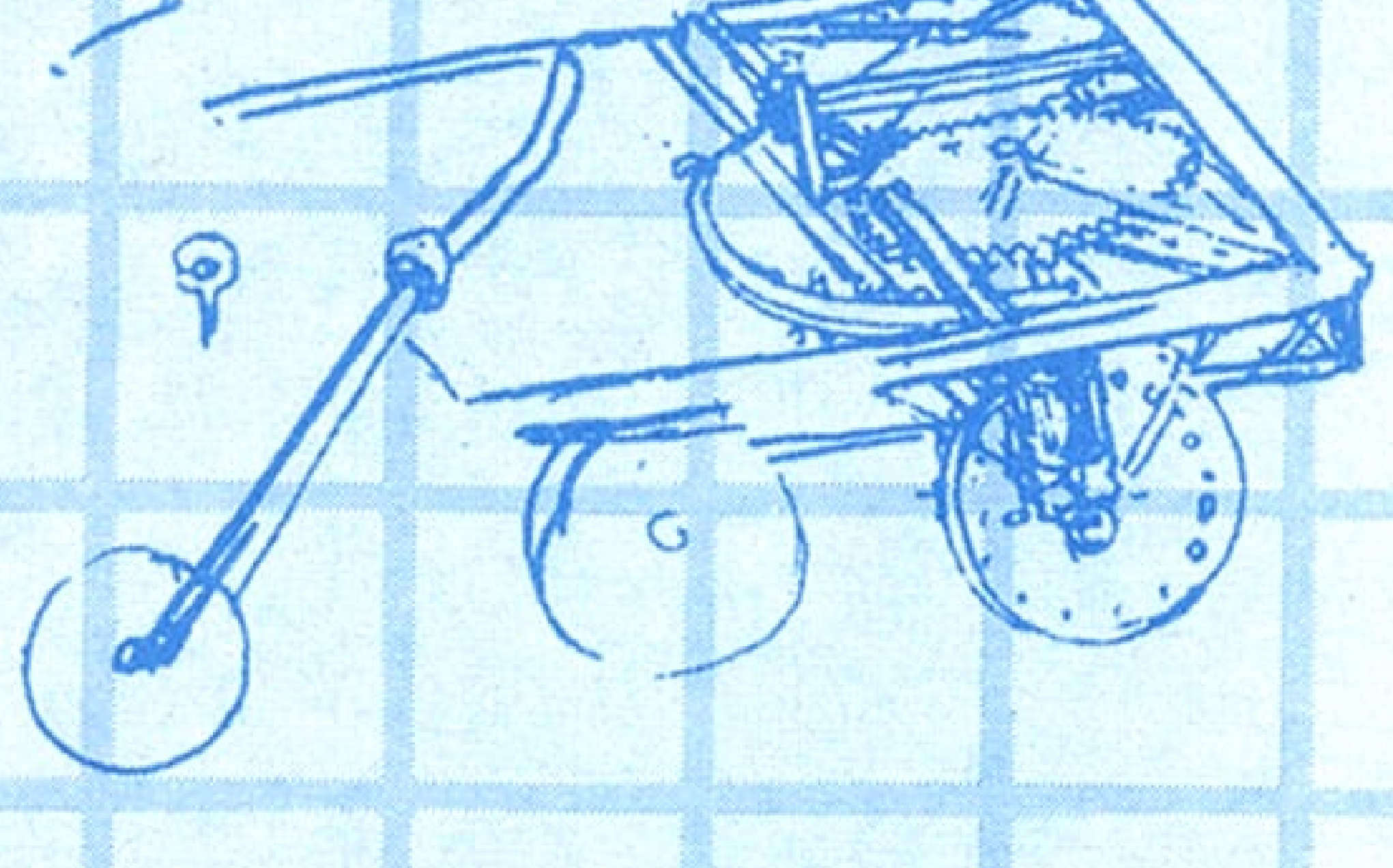
図 4.27　XFWRITE プログラムの実行例

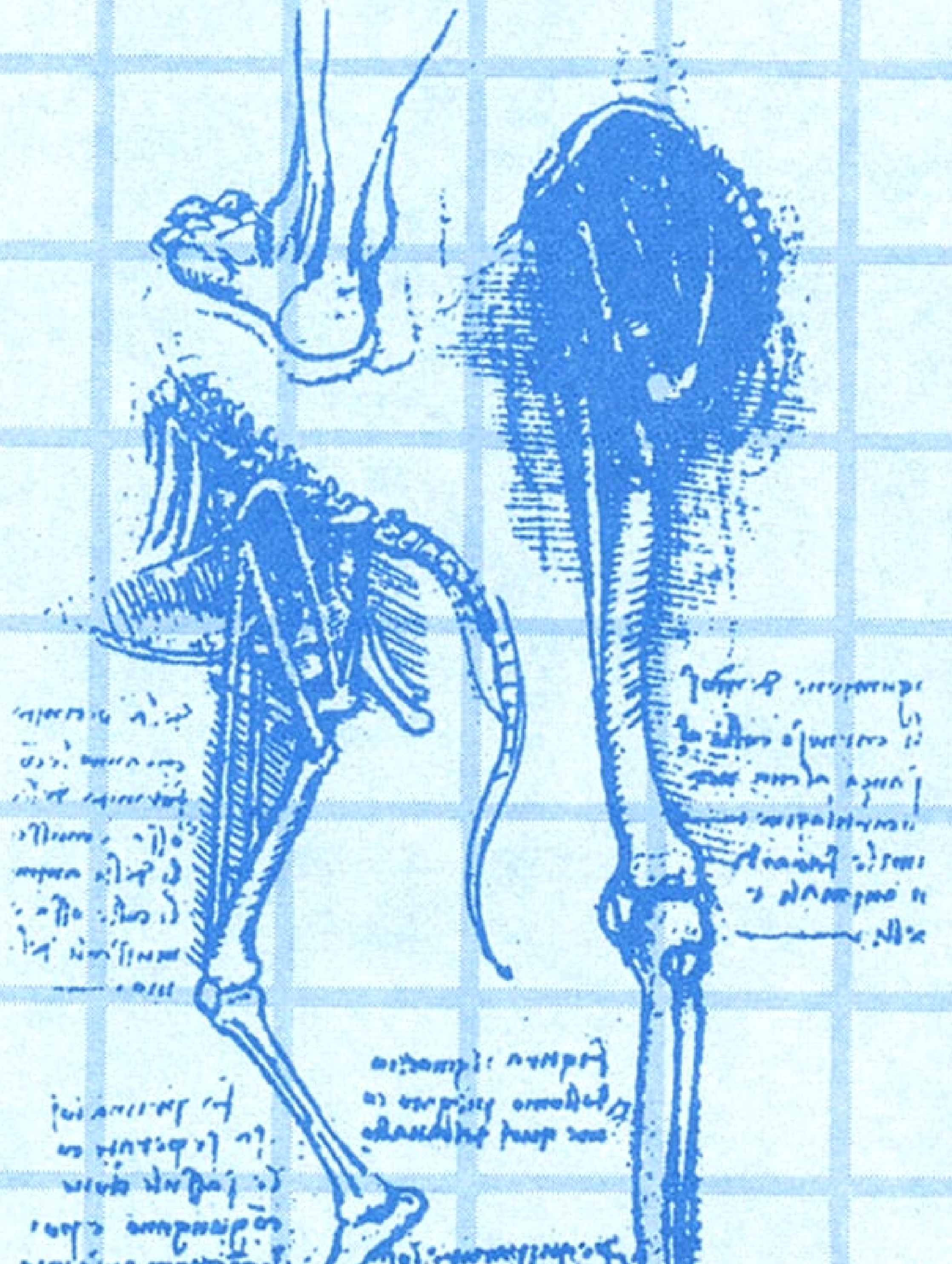
5章

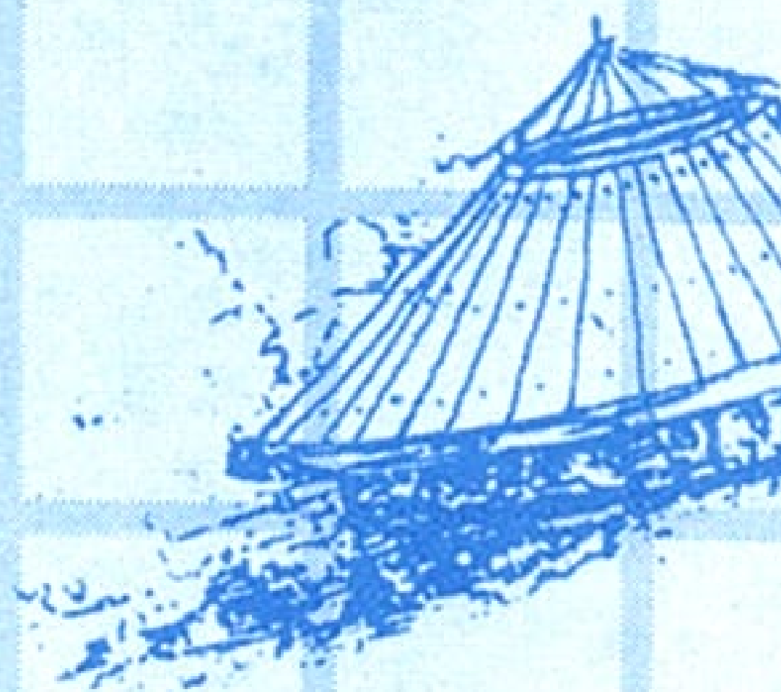
アセンブラによるソフトウェア開発











これまでに私たちは、1、2、4章などで、いくつかの簡単なプログラムをアセンブラで作成しました。小さなソースプログラムを書き、アセンブルを行い、実行可能なオブジェクトプログラムを作る手順は、いちおう体験しましたが、それらのプログラムは非常に小さくて、MS-DOSの標準マクロアセンブラ「MASM」の誇るマクロ機能をはじめとする、豊富で強力な機能はほとんど使われませんでした。

MASMは、リロケータブル・マクロアセンブラと呼ばれるアセンブラです。この形式のアセンブラが備えているリロケート機能とマクロ機能、そしてこの2つの機能の上に成り立つモジュール別プログラミングとは何でしょう。そしてリンクとは、ライブラリとは…。これらはいずれも、本格的にソフトウェアを開発するための最低限必要な知識です。本章では、これらの機能を実例を交えながら順に解き明かしていきます。また、これらの機能を使ったアセンブリ・ソースファイルの書き方や、デバッガ、それにライブラリマネージャの使い方などについても、その基礎知識を解説しましょう。

5.1　マクロアセンブラとは

MS-DOSマシンのCPUは、8086またはそれとコンパチブルのCPUです[*1]。CPUは、メモリ上にロードされている機械語と呼ばれる数値の羅列(マシン語プログラムの形態はそのようなもの)を、順次CPU内に取り込み、それを解釈して実行していく、プログラム実行処理の中枢です[*2]。MS-DOSマシン上で実行するプログラムを作るためのプログラミング言語は、BASIC、C、Pascalをはじめ、FORTRAN、COBOL、それに人工知能関係のものなど、かなり豊富にそろっていますが、どのようなプログラミング言語であっても、そこから最終的に作り出されるプログラムの形態は、みな同じ形態の実行可能なマシン語プログラムです。

高級言語の場合は、ソースファイルをコンパイルすることなどにより、実行可能なマシン語が生成されますが、その生成はそれぞれのコンパイラに委ねられています。そのため、効率のよい、きめの細かいプログラミングを行いたい場合には問題になります。そこで、プログラマーが任意のマシン語コードを自由に生成できる言語がすべてのコンピュータの最も基本的な言語として、それぞれに用意されています。それが「アセンブリ言語」です。

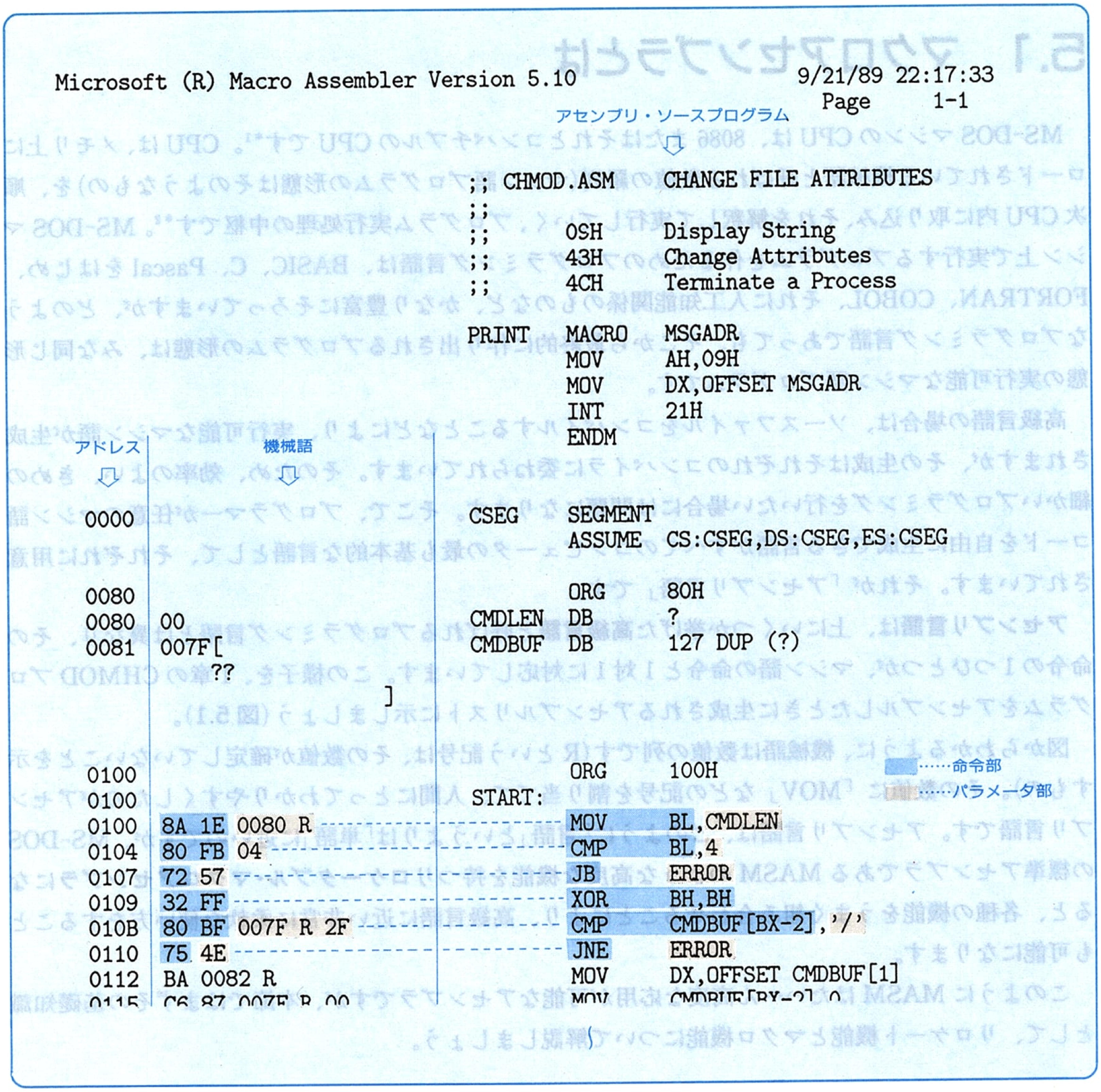
アセンブリ言語は、上にいくつか挙げた**高級言語**と呼ばれるプログラミング言語とは異なり、その命令の1つひとつが、マシン語の命令と1対1に対応しています。この様子を、1章のCHMODプログラムをアセンブルしたときに生成されるアセンブルリストに示しましょう(**図5.1**)。

図からわかるように、機械語は数値の列です(Rという記号は、その数値が確定していないことを示すもの)。その数値に「MOV」などの記号を割り当てて、人間にとってわかりやすくしたのがアセンブリ言語です。アセンブリ言語は、このように「言語」というよりは「単語」に近いのですが、MS-DOSの標準アセンブラであるMASMのような高度な機能を持つ**リロケータブル・マクロアセンブラ**になると、各種の機能をうまく組み合わせることにより、高級言語に近い非常に柔軟な使い方をすることも可能になります。

このようにMASMはたいへん高度な応用が可能なアセンブラですが、本節ではまずその基礎知識として、リロケート機能とマクロ機能について解説しましょう。

[*1] 80186、80286、80386、80486、V30などは、すべて8086とコンパチブルのCPUで、8086に比べて大幅に機能拡張されているが、MS-DOSでは8086としての機能しか利用していない。

[*2] マシン語やアセンブラについての予備知識がない方は、『はじめて読む8086』『はじめて読むMASM』(ともにアスキー出版局発行)を読まれることをお勧めします。マシン語やアセンブラの最も基礎的なことをやさしく説いています。



```
 Microsoft (R) Macro Assembler Version 5.10                 9/21/89 22:17:33
                                                             Page     1-1

                                       ;; CHMOD.ASM    CHANGE FILE ATTRIBUTES
                                       ;;
                                       ;;      09H      Display String
                                       ;;      43H      Change Attributes
                                       ;;      4CH      Terminate a Process

                                       PRINT   MACRO    MSGADR
                                               MOV      AH,09H
                                               MOV      DX,OFFSET MSGADR
                                               INT      21H
                                               ENDM

  0000                                 CSEG    SEGMENT
                                               ASSUME   CS:CSEG,DS:CSEG,ES:CSEG

  0080                                         ORG      80H
  0080  00                             CMDLEN  DB       ?
  0081  007F[                          CMDBUF  DB       127 DUP (?)
            ??
                       ]

  0100                                         ORG      100H
  0100                                 START:
  0100  8A 1E 0080 R                           MOV      BL,CMDLEN
  0104  80 FB 04                               CMP      BL,4
  0107  72 57                                  JB       ERROR
  0109  32 FF                                  XOR      BH,BH
  010B  80 BF 007F R 2F                        CMP      CMDBUF[BX-2],'/'
  0110  75 4E                                  JNE      ERROR
  0112  BA 0082 R                              MOV      DX,OFFSET CMDBUF[1]
  0115  C6 87 007F R 00                        MOV      CMDBUF[BX-2],0
```

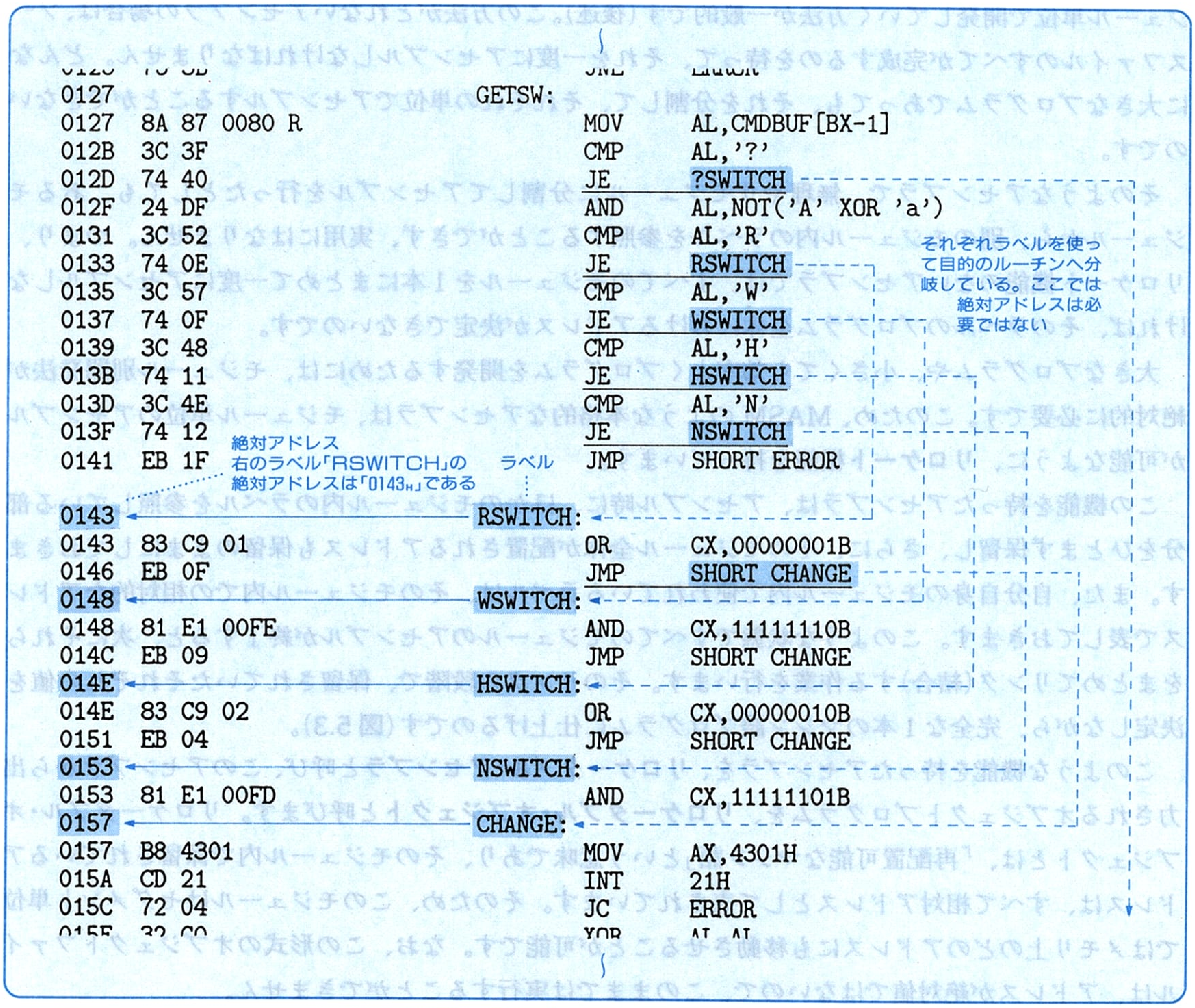
図 5.1　アセンブリ言語とマシン語(CHMOD プログラムのリスティングファイルの一部)

■ リロケート機能

アセンブラの最も基本的な機能には、アセンブリ・ソースファイル(以降は単にソースファイルと記す)の各命令を、1対1でマシン語のコードに変換する処理と、ラベルの処理があります。ラベルは、図 5.1 に示したアセンブルリストにも見えていますが、プログラムの中の特定のアドレスを、その絶対値でなく名前(ラベル)で表す(参照する)ためのものです。

CPUは、メモリ上の命令を1つ取り出して実行し、その実行が終わると、次のアドレスに格納されている命令を取り出して実行する処理を繰り返します。ところがCPU命令の中には、実行がアドレス順ではなく、任意のアドレスへ飛んで、そこの命令から実行を続ける、**分岐命令**と呼ばれるものがあります。ジャンプ命令やコール命令などがそれにあたります。

しかし、分岐先のアドレスを数値で指定するのは、なかなかやっかいな仕事です。分岐先のアドレスは、現在のアドレスと分岐先のアドレスとの間にあるソースファイルのステップ数や、その命令の種類(つまり、その間のプログラムのバイト数)によって決まりますが、このバイト数を人間が数えるとなるとたいへんです。しかしアセンブラは、このバイト数を、分岐先に付けられたラベルと、分岐先として指定した同じラベルから、その数値を自動的に計算してくれます(**図5.2**参照)。



```
0127                          GETSW:
0127  8A 87 0080 R                   MOV   AL,CMDBUF[BX-1]
012B  3C 3F                          CMP   AL,'?'
012D  74 40                          JE    ?SWITCH
012F  24 DF                          AND   AL,NOT('A' XOR 'a')
0131  3C 52                          CMP   AL,'R'
0133  74 0E                          JE    RSWITCH
0135  3C 57                          CMP   AL,'W'
0137  74 0F                          JE    WSWITCH
0139  3C 48                          CMP   AL,'H'
013B  74 11                          JE    HSWITCH
013D  3C 4E                          CMP   AL,'N'
013F  74 12                          JE    NSWITCH
0141  EB 1F                          JMP   SHORT ERROR
0143                          RSWITCH:
0143  83 C9 01                       OR    CX,00000001B
0146  EB 0F                          JMP   SHORT CHANGE
0148                          WSWITCH:
0148  81 E1 00FE                     AND   CX,11111110B
014C  EB 09                          JMP   SHORT CHANGE
014E                          HSWITCH:
014E  83 C9 02                       OR    CX,00000010B
0151  EB 04                          JMP   SHORT CHANGE
0153                          NSWITCH:
0153  81 E1 00FD                     AND   CX,11111101B
0157                          CHANGE:
0157  B8 4301                        MOV   AX,4301H
015A  CD 21                          INT   21H
015C  72 04                          JC    ERROR
```

それぞれラベルを使って目的のルーチンへ分岐している。ここでは絶対アドレスは必要ではない
絶対アドレス
右のラベル「RSWITCH」の絶対アドレスは「0143_H」である
ラベル

図5.2　ラベルの機能(CHMODプログラムのリスティングファイルの一部)

また、アセンブラのその他の基本機能としては、アドレスや定数、変数などの数値を名前で表す**シンボル**という概念や、プログラムの先頭アドレスを指定する命令、メモリ領域を確保する命令といったような**擬似命令**(CPUに対する命令ではなく、アセンブラの実行を制御する命令)などがあります。

リロケートやマクロ機能を持たない機能の低いアセンブラ(たとえばCP／M-80やCP／M-86に付属のアセンブラ)でも、およそ以上のような機能は持っています。これだけの機能でも、小規模なプログラムであれば開発することが可能ですが、大きなプログラムの開発や開発効率を考えると、いろいろな問題が出てきます。

モジュール

大きなプログラムを開発する場合は、1つのプログラムをいくつかのモジュールに分け、そのモジュール単位で開発していく方法が一般的です(後述)。この方法がとれないアセンブラの場合は、ソースファイルのすべてが完成するのを待って、それを一度にアセンブルしなければなりません。どんなに大きなプログラムであっても、それを分割して、それぞれの単位でアセンブルすることができないのです。

そのようなアセンブラで、無理やりモジュールに分割してアセンブルを行ったとしても、あるモジュールから、別のモジュール内のラベルを参照することができず、実用にはなりません。つまり、リロケート機能のないアセンブラでは、すべてのモジュールを1本にまとめて一度にアセンブルしなければ、そのラベルのプログラム全体におけるアドレスが決定できないのです。

大きなプログラムや、小さくても効率よくプログラムを開発するためには、モジュール別開発法が絶対的に必要です。このため、MASMのような本格的なアセンブラは、モジュール単位のアセンブルが可能なように、**リロケート機能**を持っています。

この機能を持ったアセンブラは、アセンブル時に、ほかのモジュール内のラベルを参照している部分をひとまず保留し、さらに、そのモジュール全体が配置されるアドレスも保留のままにしておきます。また、自分自身のモジュール内で使われているラベルは、そのモジュール内での相対的なアドレスで表しておきます。このような状態ですべてのモジュールのアセンブルが終了すると、次にそれらをまとめてリンク(結合)する作業を行います。そのリンクの段階で、保留されていたそれぞれの値を決定しながら、完全な1本のマシン語プログラムに仕上げるのです(**図5.3**)。

このような機能を持ったアセンブラを、**リロケータブル・アセンブラ**と呼び、このアセンブラから出力されるオブジェクトプログラムを、**リロケータブル・オブジェクト**と呼びます。リロケータブル・オブジェクトとは、「再配置可能なマシン語」という意味であり、そのモジュール内で保留されているアドレスは、すべて相対アドレスとして表されています。そのため、このモジュールはセグメント単位ではメモリ上のどのアドレスにも移動させることが可能です。なお、この形式のオブジェクトファイルは、アドレスが絶対値ではないので、このままでは実行することができません。

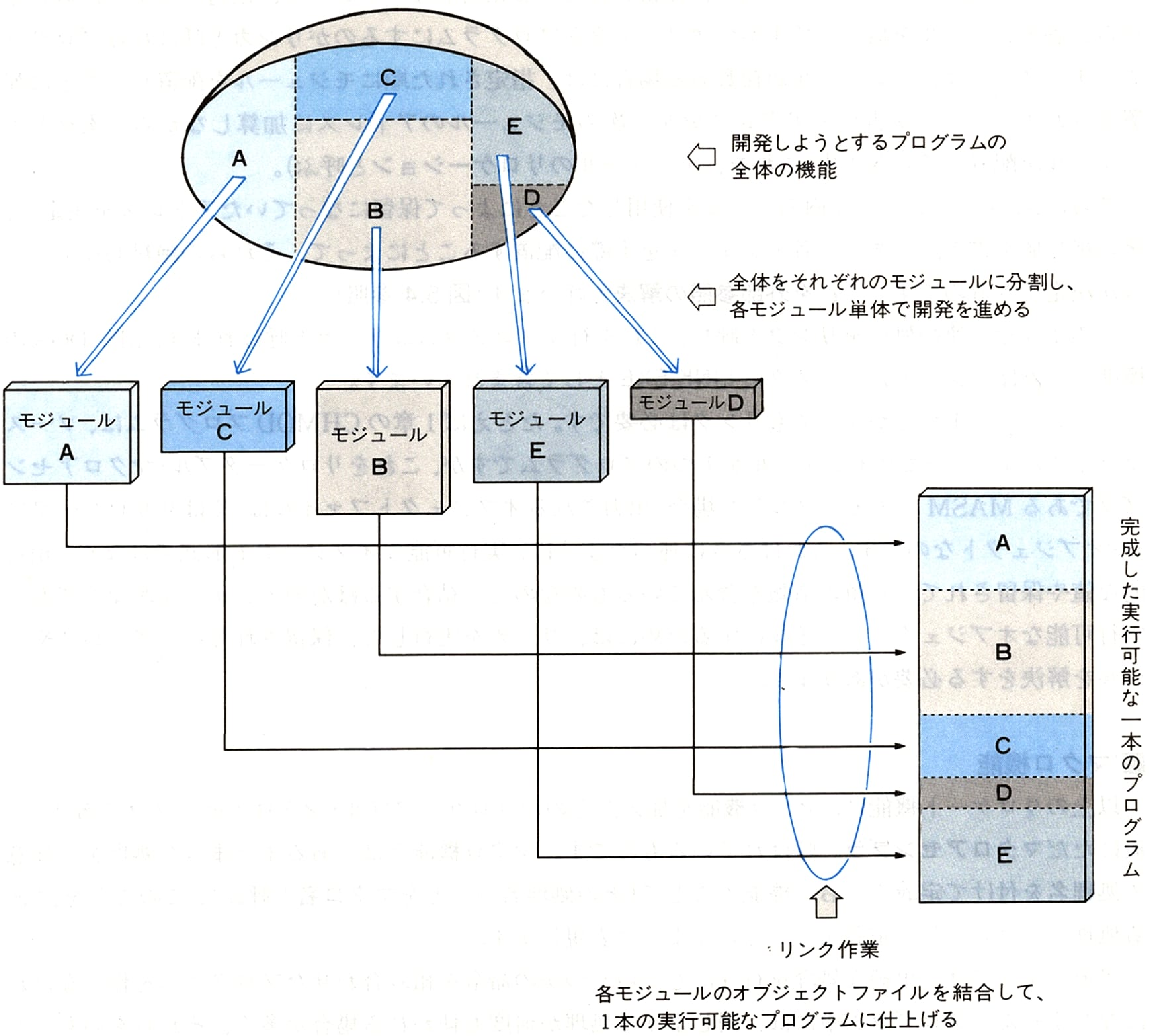

A
C
E
B
D
開発しようとするプログラムの
全体の機能
全体をそれぞれのモジュールに分割し、
各モジュール単体で開発を進める
モジュール
A
モジュール
C
モジュール
B
モジュール
E
モジュールD
A
B
C
D
E
完成した実行可能な一本のプログラム
リンク作業
各モジュールのオブジェクトファイルを結合して、
1本の実行可能なプログラムに仕上げる

図 5.3　モジュール別アセンブルの概念

リンカ

アセンブル済みの各モジュール内の、保留されている相対的なアドレスを、絶対アドレスに直しながら、各モジュールを結合して1本のオブジェクトプログラムにするのが**リンカ**と呼ばれるプログラムです。リンカは、モジュールが複数ある場合には、指定された順にモジュールを配置し、さきに配置されたモジュールが占めるアドレス分を、次のモジュールのアドレスに加算しながら、次々とモジュールを配置していきます(これを、モジュールの**リロケーション**と呼ぶ)。

さらに、ほかのモジュール内のラベルを使用したことによって保留になっていたアドレスを決定し、その絶対値を書き込みます。各モジュールを実際に配置することによって、それらの絶対的なアドレスが決定できるのです。これを**外部参照の解決**と呼びます(**図5.4**参照)。

このような一連の処理を**リンク**と呼び、これを行うプログラムはリンカと呼ばれます。MS-DOSの標準リンカは、システムディスクに**LINK.EXE**として含まれています。

モジュールが1つしかなくてもリンクは必要です。たとえば1章のCHMODプログラムは、ソースファイルが1つ、つまりモジュールが1つのプログラムですが、これをリロケータブル・マクロアセンブラであるMASMでアセンブルした場合、出力されるオブジェクトファイルは、やはりリロケータブル・オブジェクトなのです。これはさきに述べたように、実行可能なオブジェクト形式ではなく、相対的な値や保留されている値の情報を含んでいるものなので、結合するほかのモジュールがなくても、実行可能なオブジェクトファイルにするためには、リンカを実行して、保留されているアドレスやラベルを解決をする必要があります。

■ マクロ機能

以上のリロケート機能に、マクロ機能を加えたものがリロケータブル・マクロアセンブラであり、一般にただ**マクロアセンブラ**と呼ばれているものです。マクロ機能とは、あるまとまった処理を、任意の処理名を付けて定義しておく機能のことで(その処理名のことを**マクロ名**と呼ぶ)、このことを、「ある処理をマクロとして定義する」というように表現します。

アセンブラでは、単純な処理を行うにも、いくつかの命令を組み合わせたプログラムを作らなければなりません。プログラム中には、同じような処理が何度も使われる場合が多く、それらをいちいち書いていたのではプログラムを作成する効率やプログラムの見通しが悪くなります。

そこで、ある処理(いくつかのCPU命令から成るプログラム)を1つの命令として定義し、定義された処理は1つの命令として取り扱えるようにします(これを**マクロ定義**と呼ぶ)。定義したマクロは、プログラム上では、あたかも1つの命令であるかのように書くことが可能です。このように、定義したマクロをプログラム上で使うこと(記述すること)を**マクロ呼び出し**といいますが、この機能は、10桁の電話番号を3桁で表してしまう短縮ダイヤルの登録とその呼び出しによく似ています(**図5.5**参照)。

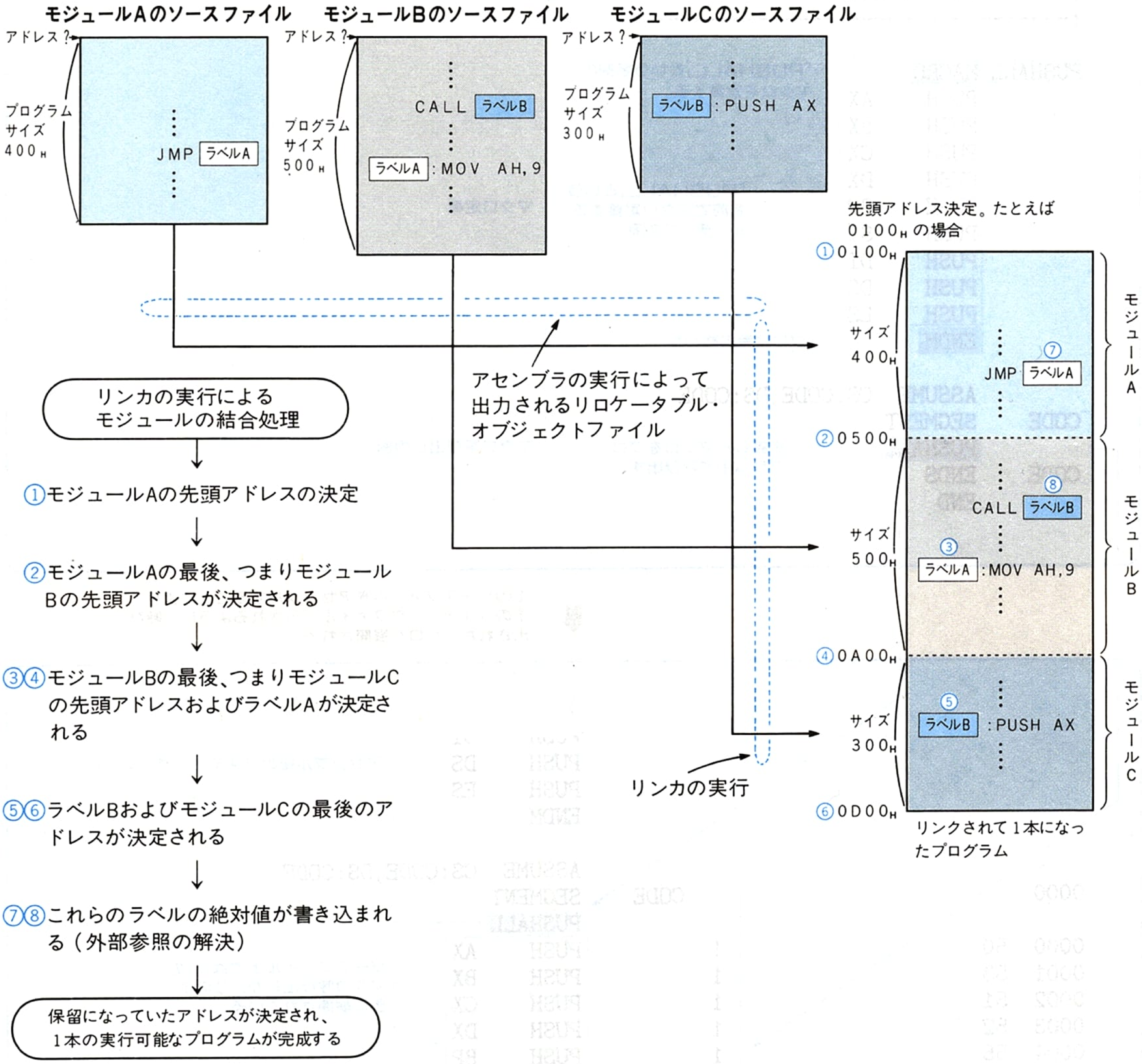
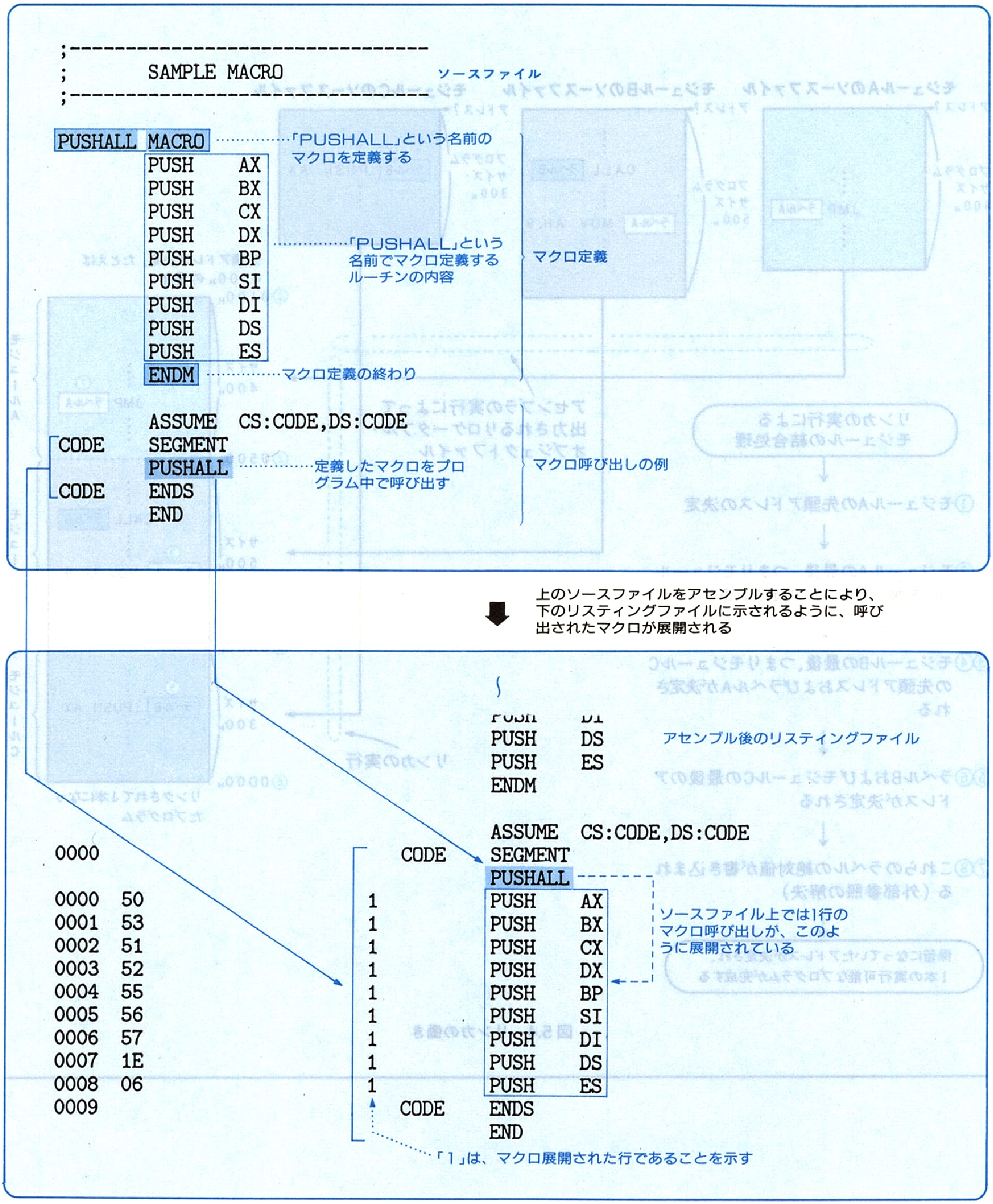

モジュールAのソースファイル
モジュールBのソースファイル
モジュールCのソースファイル
アドレス？
プログラムサイズ 400H
JMP ラベルA
アドレス？
CALL ラベルB
プログラムサイズ 500H
ラベルA：MOV AH，9
アドレス？
プログラムサイズ 300H
ラベルB：PUSH AX
先頭アドレス決定。たとえば 0100H の場合
① 0100H
サイズ 400H
⑦ JMP ラベルA
② 0500H
⑧ CALL ラベルB
サイズ 500H
③ ラベルA：MOV AH，9
④ 0A00H
⑤ ラベルB：PUSH AX
サイズ 300H
⑥ 0D00H
モジュールA
モジュールB
モジュールC
リンクされて1本になったプログラム
アセンブラの実行によって出力されるリロケータブル・オブジェクトファイル
リンカの実行
リンカの実行によるモジュールの結合処理
①モジュールAの先頭アドレスの決定
②モジュールAの最後、つまりモジュールBの先頭アドレスが決定される
③④モジュールBの最後、つまりモジュールCの先頭アドレスおよびラベルAが決定される
⑤⑥ラベルBおよびモジュールCの最後のアドレスが決定される
⑦⑧これらのラベルの絶対値が書き込まれる（外部参照の解決）
保留になっていたアドレスが決定され、1本の実行可能なプログラムが完成する

図 5.4　リンカの働き



```
;---------------------------------
;        SAMPLE MACRO
;---------------------------------

PUSHALL MACRO
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    BP
        PUSH    SI
        PUSH    DI
        PUSH    DS
        PUSH    ES
        ENDM

        ASSUME  CS:CODE,DS:CODE
CODE    SEGMENT
        PUSHALL
CODE    ENDS
        END
```

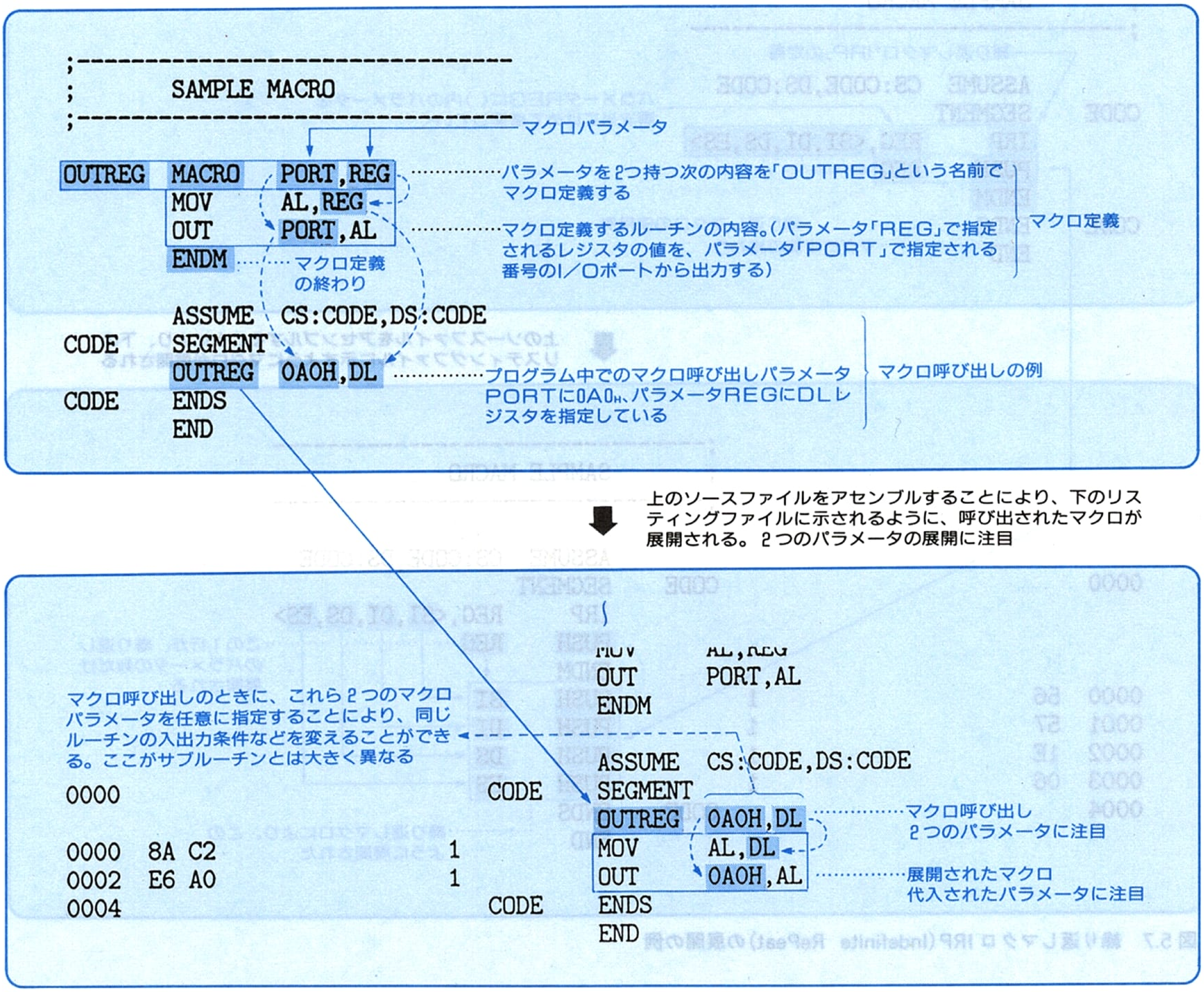
```
                             PUSH    DS
                             PUSH    ES
                             ENDM

                             ASSUME  CS:CODE,DS:CODE
0000                  CODE   SEGMENT
                             PUSHALL
0000  50           1         PUSH    AX
0001  53           1         PUSH    BX
0002  51           1         PUSH    CX
0003  52           1         PUSH    DX
0004  55           1         PUSH    BP
0005  56           1         PUSH    SI
0006  57           1         PUSH    DI
0007  1E           1         PUSH    DS
0008  06           1         PUSH    ES
0009                  CODE   ENDS
                             END
```

ソースファイル
「PUSHALL」という名前のマクロを定義する
「PUSHALL」という名前でマクロ定義するルーチンの内容
マクロ定義
マクロ定義の終わり
定義したマクロをプログラム中で呼び出す
マクロ呼び出しの例
上のソースファイルをアセンブルすることにより、下のリスティングファイルに示されるように、呼び出されたマクロが展開される
アセンブル後のリスティングファイル
ソースファイル上では1行のマクロ呼び出しが、このように展開されている
「1」は、マクロ展開された行であることを示す

図 5.5　マクロ呼び出しとマクロ呼び出しが展開された簡単な例

しかし、これだけの機能ならサブルーチンコールと同じではないか、と思った方もいるでしょう。確かに、ある処理を1つの命令として扱うだけであれば、機能的にはサブルーチンとあまり変わりません。しかしマクロ機能には、サブルーチンにはない強力な機能があるのです。

プログラムを作成していると、ほとんど同じ処理なのに、ある一部分が異なるだけの処理がいくつも出てくる場合があります。マクロ機能は、その異なる部分のみをパラメータ扱いにすることが可能で、これによって一部分が異なるだけの同じような処理を、1つのマクロ定義で表すことができます。たとえば、使用するレジスタが異なるだけで、ほかは同じプログラムである処理の場合には、その変化するレジスタ名をパラメータとして任意の名前にしておき、それらの処理を1つのマクロとして定義しておきます。そしてプログラム上でそのマクロを呼び出すときに、実際のレジスタ名を**マクロパラメータ**として指定するのです(図5.6参照)。



```
;---------------------------------
;         SAMPLE MACRO
;---------------------------------

OUTREG  MACRO    PORT,REG
        MOV      AL,REG
        OUT      PORT,AL
        ENDM

        ASSUME   CS:CODE,DS:CODE
CODE    SEGMENT
        OUTREG   0A0H,DL
CODE    ENDS
        END
```

マクロパラメータ

パラメータを2つ持つ次の内容を「OUTREG」という名前でマクロ定義する

マクロ定義するルーチンの内容。(パラメータ「REG」で指定されるレジスタの値を、パラメータ「PORT」で指定される番号のI/Oポートから出力する)

マクロ定義

マクロ定義の終わり

プログラム中でのマクロ呼び出しパラメータPORTに0A0H、パラメータREGにDLレジスタを指定している

マクロ呼び出しの例

上のソースファイルをアセンブルすることにより、下のリスティングファイルに示されるように、呼び出されたマクロが展開される。2つのパラメータの展開に注目

```
                                    MOV      AL,REG
                                    OUT      PORT,AL
                                    ENDM

                                    ASSUME   CS:CODE,DS:CODE
0000                        CODE    SEGMENT
                                    OUTREG   0A0H,DL
0000   8A C2            1           MOV      AL,DL
0002   E6 A0            1           OUT      0A0H,AL
0004                        CODE    ENDS
                                    END
```

マクロ呼び出しのときに、これら2つのマクロパラメータを任意に指定することにより、同じルーチンの入出力条件などを変えることができる。ここがサブルーチンとは大きく異なる

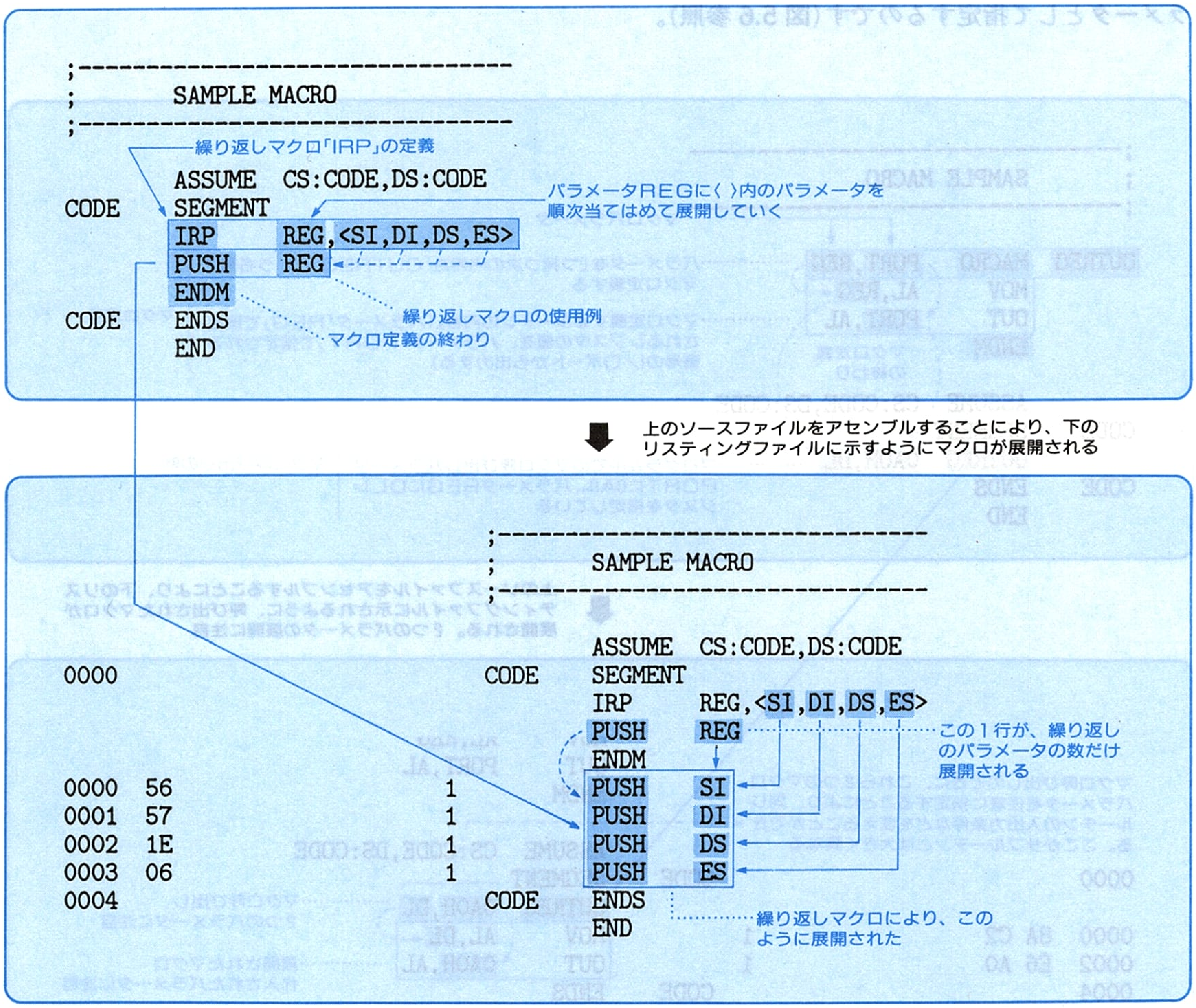
マクロ呼び出し
2つのパラメータに注目

展開されたマクロ
代入されたパラメータに注目

図5.6　マクロパラメータを使ったマクロ定義とマクロ呼び出しの例

1章で作成したCHMODプログラム（リスト1.1参照）の冒頭部で、メッセージの置かれているアドレス「MSGADR」をマクロパラメータとした文字列出力のマクロ「PRINT」を定義しています。参照してください。

マクロ機能には、以上のようなユーザーが定義するマクロのほかに、**システムマクロ**（最初からアセンブラに組み込まれているマクロ機能）と呼ばれる**リピート擬似命令**があります。これは、同じデータを繰り返し定義したり、似たようなプログラムをいくつも並べたい場合に使うもので、これらをさきのユーザー定義マクロと組み合わせることにより、実に柔軟にプログラムを記述することが可能になります（図5.7参照）。



```
;-----------------------------------
;       SAMPLE MACRO
;-----------------------------------
        ASSUME  CS:CODE,DS:CODE
CODE    SEGMENT
        IRP     REG,<SI,DI,DS,ES>
        PUSH    REG
        ENDM
CODE    ENDS
        END
```

繰り返しマクロ「IRP」の定義

パラメータREGに〈 〉内のパラメータを順次当てはめて展開していく

繰り返しマクロの使用例

マクロ定義の終わり

上のソースファイルをアセンブルすることにより、下のリスティングファイルに示すようにマクロが展開される

```
                                  ;-----------------------------------
                                  ;       SAMPLE MACRO
                                  ;-----------------------------------
                                          ASSUME  CS:CODE,DS:CODE
0000                              CODE    SEGMENT
                                          IRP     REG,<SI,DI,DS,ES>
                                          PUSH    REG
                                          ENDM
0000    56                    1           PUSH    SI
0001    57                    1           PUSH    DI
0002    1E                    1           PUSH    DS
0003    06                    1           PUSH    ES
0004                              CODE    ENDS
                                          END
```

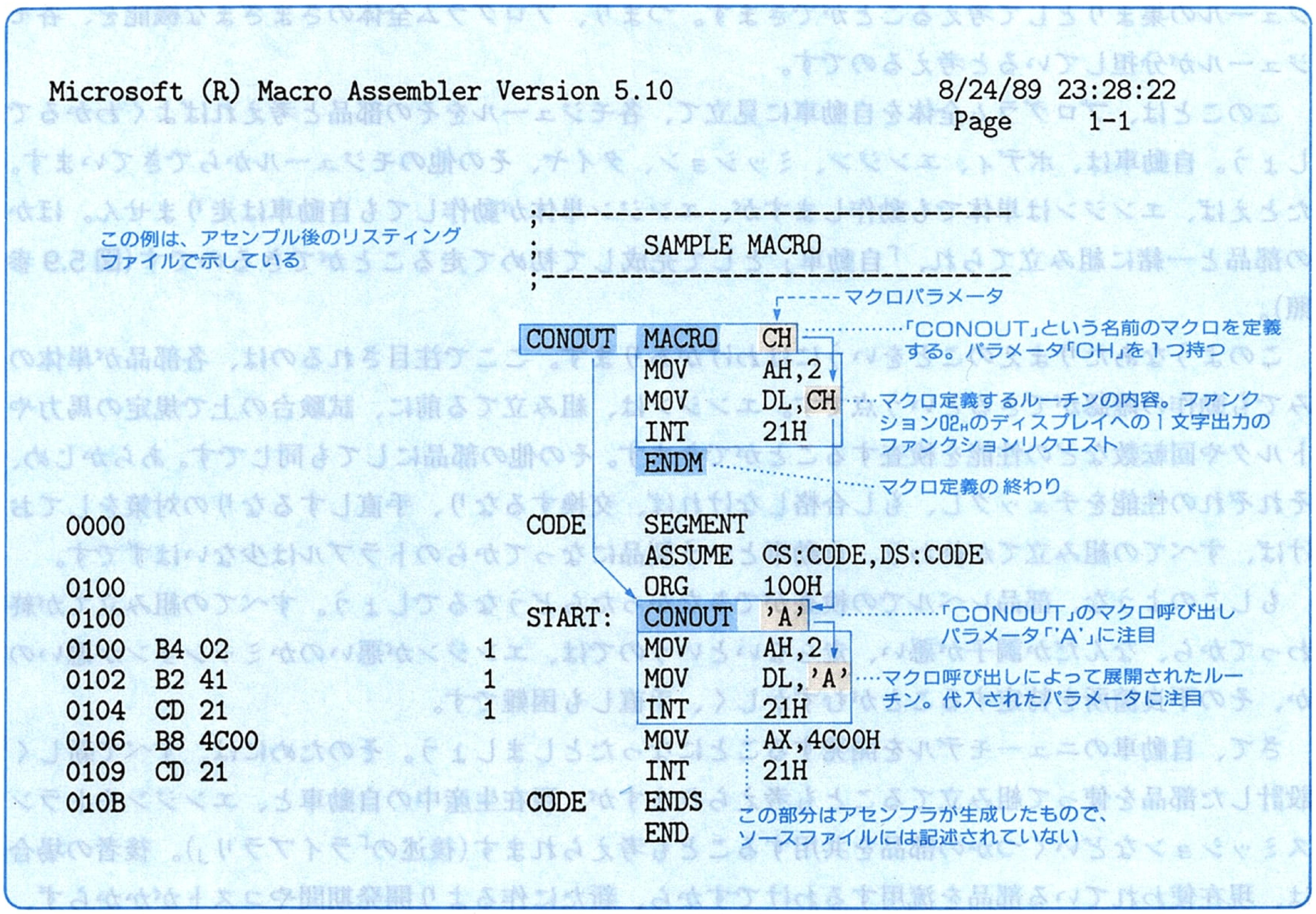
この1行が、繰り返しのパラメータの数だけ展開される

繰り返しマクロにより、このように展開された

図5.7　繰り返しマクロIRP（Indefinite　RePeat）の展開の例

さきほども少し触れましたが、マクロ機能はサブルーチンの一種のように思われがちです。ところが、今まで見てきたことからもわかるように、単機能のサブルーチンとはだいぶ異なった概念を持つ機能であり、非常に柔軟な処理が可能になります。マクロ機能は、CPUの命令セットを拡張する(ユーザーが新たに作る)ようなもの、と考えてもよいでしょう。うまく使えばプログラム全体をたいへん能率的に記述することが可能であり、同時にプログラムの構造を、見通しのよい、わかりやすいものにすることができます。マクロ機能を利用するメリットは、プログラムを記述する能率を上げるばかりではありません。たとえ1度しか使わないような処理でも、それをマクロ定義することにより、プログラムの見通しがずっとよくなることがあります。わかりやすく読みやすいプログラム、これはプログラミング上で最も重要なことです。

たとえばシステムコールを、そのファンクションの内容を表すマクロ名で定義しておけば、通常のファンクションナンバーでコールするのに比べて、はるかにプログラムがわかりやすくなることは容易に想像できるでしょう(**図5.8**参照)。



```
 Microsoft (R) Macro Assembler Version 5.10          8/24/89 23:28:22
                                                      Page     1-1


                                 ;------------------------------
                                 ;       SAMPLE MACRO
                                 ;------------------------------

                                 CONOUT  MACRO   CH
                                         MOV     AH,2
                                         MOV     DL,CH
                                         INT     21H
                                         ENDM

  0000                           CODE    SEGMENT
                                         ASSUME  CS:CODE,DS:CODE
  0100                                   ORG     100H
  0100                           START:  CONOUT  'A'
  0100  B4 02                1           MOV     AH,2
  0102  B2 41                1           MOV     DL,'A'
  0104  CD 21                1           INT     21H
  0106  B8 4C00                          MOV     AX,4C00H
  0109  CD 21                            INT     21H
  010B                           CODE    ENDS
                                         END
```

この例は、アセンブル後のリスティングファイルで示している

マクロパラメータ

「CONOUT」という名前のマクロを定義する。パラメータ「CH」を1つ持つ

マクロ定義するルーチンの内容。ファンクション02_{H}のディスプレイへの1文字出力のファンクションリクエスト

マクロ定義の終わり

「CONOUT」のマクロ呼び出しパラメータ「'A'」に注目

マクロ呼び出しによって展開されたルーチン。代入されたパラメータに注目

この部分はアセンブラが生成したもので、ソースファイルには記述されていない

図5.8　システムコールをマクロ定義した例

本書では、マクロ機能についてはこれ以上深くは触れません。詳しくはマニュアルやほかの参考書を頼りに、実際の経験を積みながら勉強してください。じっくり研究してみると、MASM のマクロ機能は実に強力で奥が深く、本格的に使いこなせば、これがアセンブラかと思うほどのことができます。条件アセンブルなどの機能と組み合わせれば、高級言語並みの記述も可能となるでしょう。たとえば、

```
LET  A  :=  B  +  C
```

というような記述さえ可能になるのです。

■ モジュール別プログラム開発法

さきのリロケート機能のところでも少し触れましたが、ここでモジュール別プログラミングについて考えてみましょう。

モジュールとは、ある程度まとまった、独立性のある処理を行うルーチンの集まりのことをいいます。モジュールは、特定の機能をいくつかのルーチンで実現しており、プログラム全体は、これらモジュールの集まりとして考えることができます。つまり、プログラム全体のさまざまな機能を、各モジュールが分担していると考えるのです。

このことは、プログラム全体を自動車に見立て、各モジュールをその部品と考えればよくわかるでしょう。自動車は、ボディ、エンジン、ミッション、タイヤ、その他のモジュールからできています。たとえば、エンジンは単体でも動作しますが、エンジン単体が動作しても自動車は走りません。ほかの部品と一緒に組み立てられ、「自動車」として完成して初めて走ることができるのです(**図 5.9** 参照)。

このようなあたりまえのことをいうにはわけがあります。ここで注目されるのは、各部品が単体のみでも動作の確認ができるという点です。エンジンは、組み立てる前に、試験台の上で規定の馬力やトルクや回転数などの性能を検査することができます。その他の部品にしても同じです。あらかじめ、それぞれの性能をチェックし、もし合格しなければ、交換するなり、手直しするなりの対策をしておけば、すべての組み立てが終わり、自動車という製品になってからのトラブルは少ないはずです。

もしこのような、部品レベルでの検査ができなかったらどうなるでしょう。すべての組み立てが終わってから、なんだか調子が悪い、走らないというのでは、エンジンが悪いのかミッションが悪いのか、その不良箇所を特定することがむずかしく、手直しも困難です。

さて、自動車のニューモデルを開発することになったとしましょう。そのためには、すべて新しく設計した部品を使って組み立てることも考えられますが、現在生産中の自動車と、エンジンやトランスミッションなどいくつかの部品を共用することも考えられます(後述の「ライブラリ」)。後者の場合は、現在使われている部品を流用するわけですから、新たに作るより開発期間やコストがかからず、信頼性も高いという利点があります。反面、その部品の性能以上のものは作れず、またデザインもそれに合わせる必要があり、自由度が制約されます。

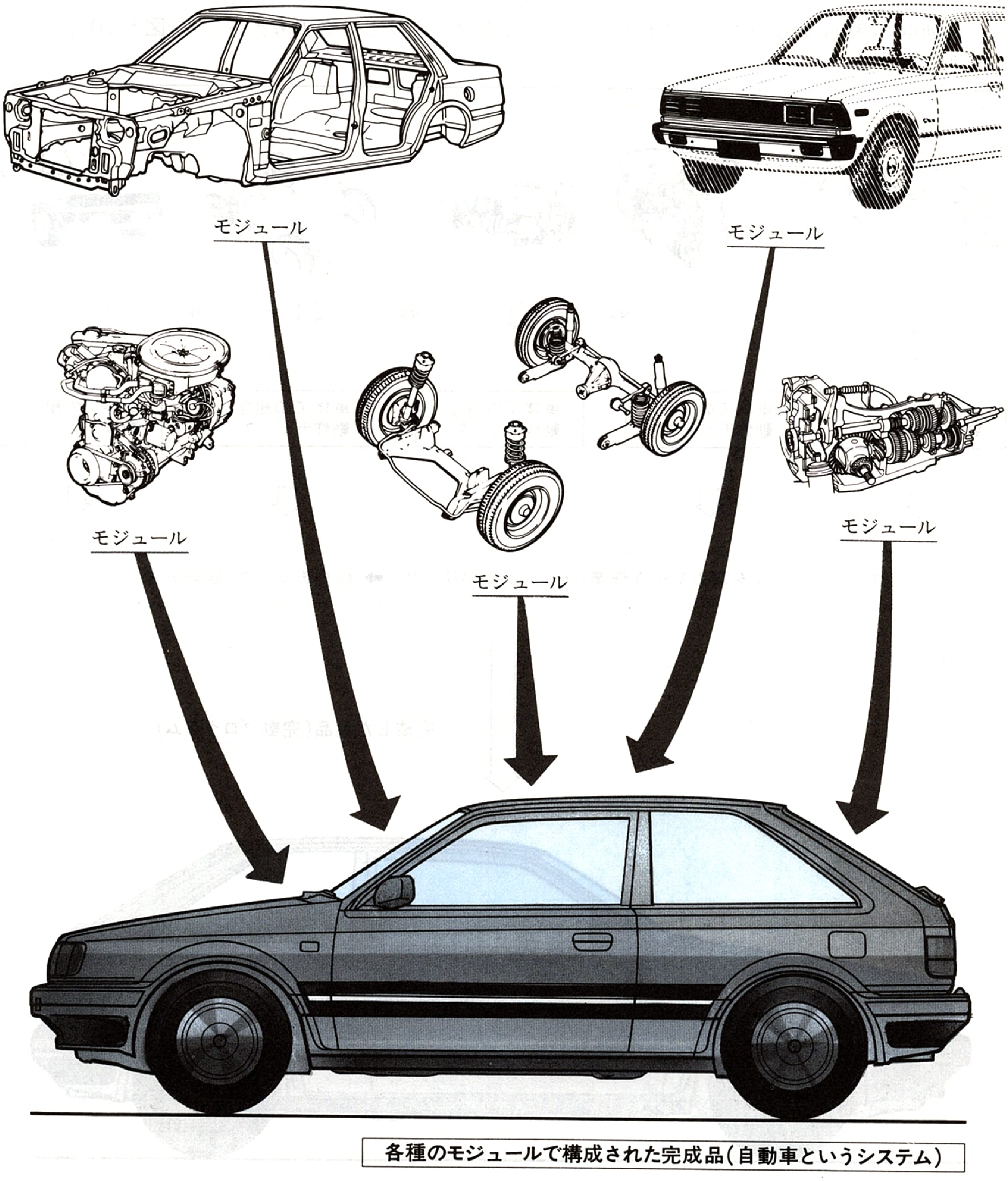

モジュール
モジュール
モジュール
モジュール
モジュール
モジュール
各種のモジュールで構成された完成品(自動車というシステム)

図5.9　自動車というシステムは、各種のモジュールから構成されている

ソフトウェア開発に関しても、自動車作りと同じようなことがいえます。プログラム全体をモジュール(部品)の集まりと考え、モジュール単位で開発、検査を行い、それらを組み合わせて１つのプログラム(自動車)を構成するのです。これがモジュール別プログラミングの考え方です(図 5.10)。

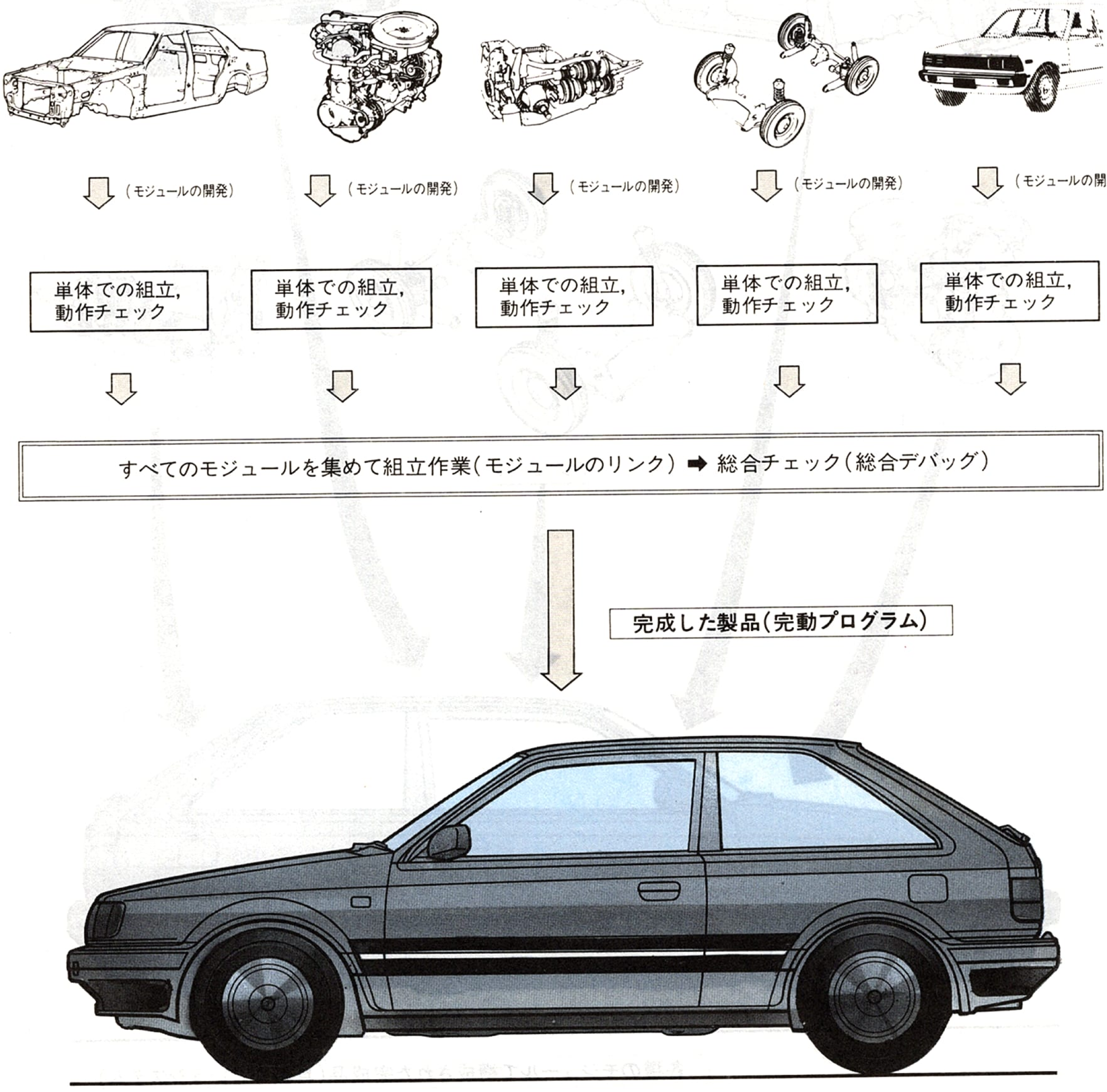

(モジュールの開発)
(モジュールの開発)
(モジュールの開発)
(モジュールの開発)
(モジュールの開発)
単体での組立，動作チェック
単体での組立，動作チェック
単体での組立，動作チェック
単体での組立，動作チェック
単体での組立，動作チェック
すべてのモジュールを集めて組立作業(モジュールのリンク) ➡ 総合チェック(総合デバッグ)
完成した製品(完動プログラム)

図 5.10　モジュール別プログラミングの仕組み

ソフトウェアの開発においては、それぞれのモジュール単体を作り、その動作を確認して入出力条件を明らかにしておけば、単なるブラックボックスとして利用することができるだけでなく、プログラミングやデバッグが非常に楽になります。

大きなプログラムを作る場合でも、それぞれのモジュール単位で開発していくことにより、作業を確実に進めることが可能です。大きなプログラムの開発は、大勢の人手によって行われることになりますが、他人の作ったモジュールでも、その入出力条件さえ明らかであれば、誰もがブラックボックスとして利用することができるのです。

また、モジュール別プログラミングは、プログラムの仕様が変更になった場合の対応がたいへん楽になります。プログラム全体の中で、関係のあるモジュールだけを変更すればよく、ほかのモジュールを変更する必要はありません。なお、モジュールの汎用性を高めるためには、それぞれのモジュールの外部条件に対する依存性を低くすることにも気を遣う必要があります。

ソフトウェアを開発するには、プログラムの読みやすさや保守性の面からも、モジュール別のプログラム開発を行うのは常識です。このモジュール別プログラミングにおける開発プログラムの基本設計は、それぞれのモジュールの構成とその組み立て方を設計することにほかならないのです。

以上の話は、モジュール別ソフトウェア開発法の話でしたが、次はソフトウェア開発におけるもう1つの重要な機能である、モジュールの共用、つまりライブラリについての話をしましょう。

ライブラリ

あるプログラムで使われたモジュールのうち、汎用性のあるものは別のプログラムでも使える可能性があります。そのようなモジュールをまとめたものを**ライブラリ**と呼び、ライブラリに登録されているモジュールは、さきに述べたマクロの呼び出しと同じように、プログラム中で呼び出すことによって、プログラム内に組み込むことができます。各種の機能を持つモジュールがたくさん集まり、ライブラリが豊富になれば、ソフトウェア開発の能率はぐんと上がります。有用なモジュールを豊富に集めたライブラリは、プログラマーの財産です。

MS-DOSには、このライブラリにモジュールを登録したり、削除したり、抜き出したりする、ライブラリを管理するのためのプログラム**LIB.EXE**が用意されています。このプログラムを**ライブラリマネージャ**、あるいは**ライブラリアン**などと呼んでいます。

ライブラリから必要なモジュールを呼び出し、プログラムに組み込むにはリンカを使います。プログラム中でライブラリ内のモジュールがEXTRN宣言されていると、リンカはライブラリを検索し、そのモジュールを呼ぶCALL命令などがあれば、通常のマクロ呼び出しの処理と同じように、指定されたモジュールを自動的にリンクする機能を持っています(**図5.11**)。(ライブラリマネージャについては、5.7でさらに解説する)。

これまで述べてきたようなモジュール別プログラミングを可能にするのが、マクロアセンブラ、リンカ、ライブラリマネージャであり、これらの機能を総合して柔軟な開発環境が実現されています。

MS-DOSにはこれらのツールが標準で付属しており、モジュール別プログラミングのための環境をサポートしています。アセンブラMASMが生成するリロケータブルオブジェクトと同じフォーマットを、標準フォーマットとして採用しているMS-DOS上の多くの高級言語などと互いにプログラムのリンクができるというのは、パーソナル・コンピュータのOSとしては画期的な、すばらしいことです。

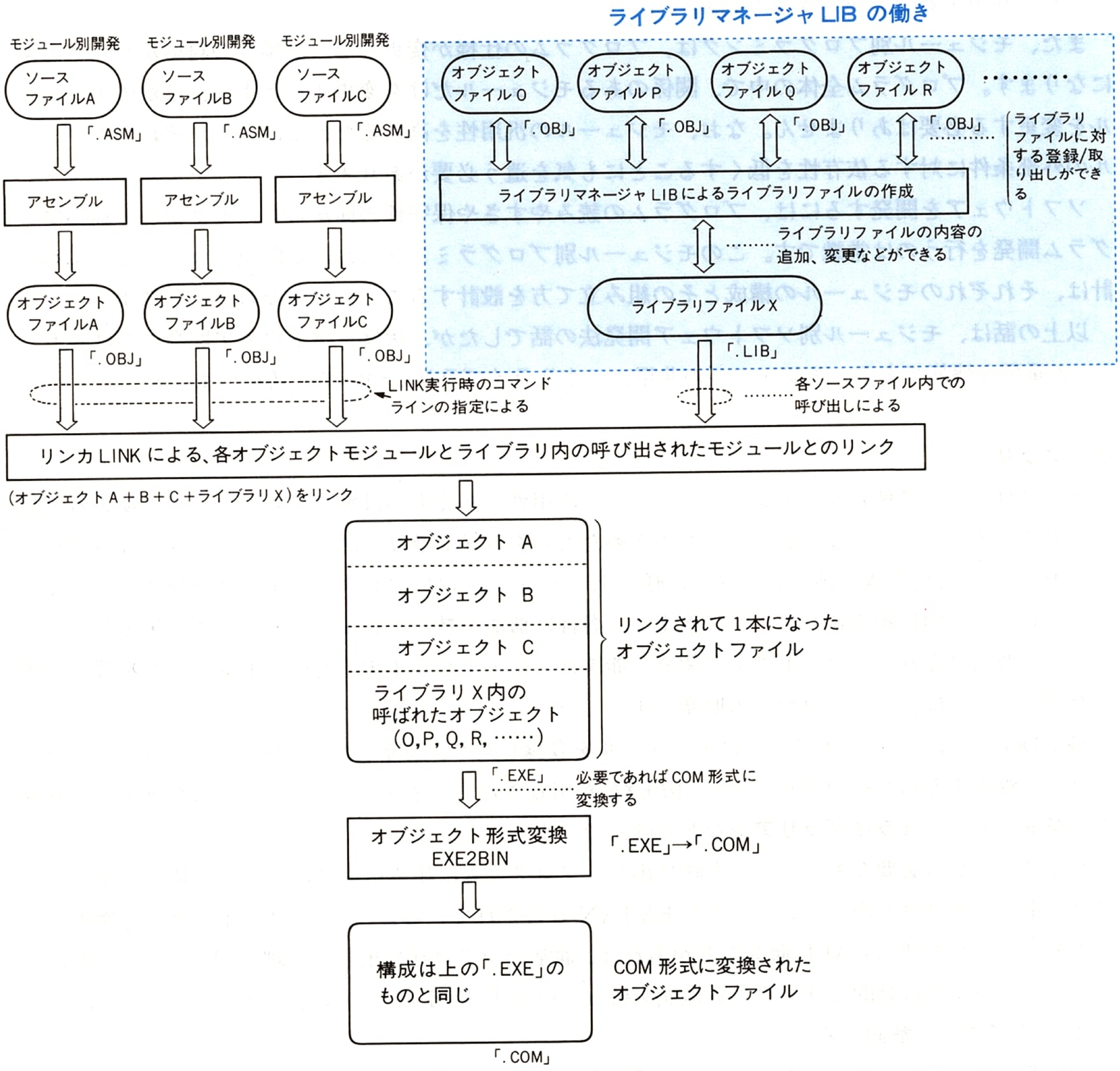

ライブラリマネージャ LIB の働き
モジュール別開発
モジュール別開発
モジュール別開発
ソース ファイルA
ソース ファイルB
ソース ファイルC
「.ASM」
「.ASM」
「.ASM」
アセンブル
アセンブル
アセンブル
オブジェクト ファイルA
オブジェクト ファイルB
オブジェクト ファイルC
「.OBJ」
「.OBJ」
「.OBJ」
オブジェクト ファイルO
オブジェクト ファイルP
オブジェクト ファイルQ
オブジェクト ファイルR
………
「.OBJ」
「.OBJ」
「.OBJ」
「.OBJ」
ライブラリファイルに対する登録/取り出しができる
ライブラリマネージャ LIBによるライブラリファイルの作成
ライブラリファイルの内容の追加、変更などができる
ライブラリファイルX
「.LIB」
LINK実行時のコマンドラインの指定による
各ソースファイル内での呼び出しによる
リンカ LINK による、各オブジェクトモジュールとライブラリ内の呼び出されたモジュールとのリンク
(オブジェクトA＋B＋C＋ライブラリX)をリンク
オブジェクト A
オブジェクト B
オブジェクト C
ライブラリX内の呼ばれたオブジェクト (O,P,Q,R,……)
リンクされて1本になったオブジェクトファイル
「.EXE」
必要であればCOM形式に変換する
オブジェクト形式変換 EXE2BIN
「.EXE」→「.COM」
構成は上の「.EXE」のものと同じ
COM形式に変換されたオブジェクトファイル
「.COM」

図5.11　アセンブラによるソフトウェア開発におけるアセンブルおよびリンク作業全体の流れ

5.2 アセンブラによるソフトウェア開発の手順

本節では、アセンブラによりソフトウェアを開発する手順を考えてみましょう。まず、発注者が要求する機能仕様に基づいてプログラムの基本設計を行いますが、その際には、プログラム全体の構成や流れを考え、モジュールへの分割とそれぞれのモジュールの機能を決定しなければなりません。この段階の設計がプログラム全体の良し悪しを決めるといってもよいでしょう。

ここでは、そのような基本設計は終わっているものとして、その次の作業である各モジュールのプログラミングから話を始めましょう。モジュールの処理内容、入力条件、出力条件を決定したら、いよいよソースファイルの作成です。次にその一連の作業を示します(図 5.12 参照)。

なお、一連の作業の繰り返し部分を自動的に行ってくれるユーティリティプログラムに MAKE があります。6 章の 6.3 で紹介していますので参照してください。

[ソースファイルの作成]

エディタを使って、モジュールごとにソースファイルを作成する。1つのモージュルが、複数のソースファイルとなってもかまわない。エディタは、MS-DOSの標準エディタEDLINでも、もっと便利な市販のスクリーンエディタでも、あるいはワードプロセッサを使ってもよい。作成するソースファイルのファイル名は、そのファイルタイプ（拡張子）を「.ASM」とする

⬇

[アセンブル]

作成した各モジュールのソースファイルを、MS-DOSの標準アセンブラMASMでアセンブルする。もし、アセンブルエラーが発生してアセンブルが成功しなかった場合は、ソースファイルを修正し、再度アセンブルする。

アセンブルが終了すると、ファイルタイプが「.OBJ」であるリロケータブル・オブジェクトファイルが生成されている。このとき、必要であれば、リスティングファイル「.LST」やクロスリファレンスファイル「.CRF」を生成することもできる。CRFファイルはバイナリファイルであるが、ユーティリティプログラムCREFを実行することにより、TYPEコマンドなどで読むことが可能なクロスリファレンス・リスティングファイルを得ることができる

⬇

(次ページに続く)

[モジュール単体の動作テスト]

各モジュールのアセンブルが成功すれば、動作テストが必要なモジュールは、それを単体でテストできるような環境を作り、仕様を満足しているかどうかを確認する。そのためには、まずリンカを実行して、モジュール単体での実行が可能オブジェクトファイルを作らなければならない。また場合によっては、テスト用の簡単なプログラムを作る必要もある

↓

[各モジュールのリンク]

それぞれのモジュールができあがり、単体でのデバッグが終わると、結合するすべてのモジュール名を指定してリンカを実行する。リンクは、MS-DOSの標準リンカLINKで行う。リンカはいくつかのOBJファイルをつなげて、1本の実行可能なEXEファイルを生成する。必要があればこのときに、セグメントのリロケーション情報についてのリスティングファイル「.MAP」を作成することもできる。

リンカ実行時に、外部のモジュールを参照しているのにそのモジュールが存在していない場合や、存在していてもそれを外部のモジュールで参照可能なように宣言しておかなかった場合、また、外部参照可能なラベルが、同じ名前で重複して定義されているモジュールが存在する場合などはエラーとなる

↓

[実行可能なオブジェクトプログラムの作成と実動テスト]

リンクが終了すると、1本の実行可能なEXE形式のオブジェクトファイルができあがる。実行可能なオブジェクトプログラムをCOM形式で作成する場合には、このあとにオブジェクト形式の変換を行う。この変換を行うプログラムがEXE2BINである。

実行可能なオブジェクトファイルができあがれば実動テストをする。これで仕様どおりに動作すれば大成功だが、たいていの場合はどこかにバグがあり、完全な動作はしない。バグがあればデバック作業を行う

↓

[デバッグ作業]

デバッグ作業は、症状をよく観察した上で、全体の流れやソースファイルをじっくり読み返し、どこに問題があるのかを考える。考えてもわからない場合は、問題となっている部分の確認や発見などにデバッガの助けを借りることができる。デバッガはプログラムの動作を、様々な面から調べることができ、プログラムを実際に動作させながら観察することが可能である。このデバッガを有効に使いこなせるかどうかは、ソフトウェア開発全体に大きな影響を与える

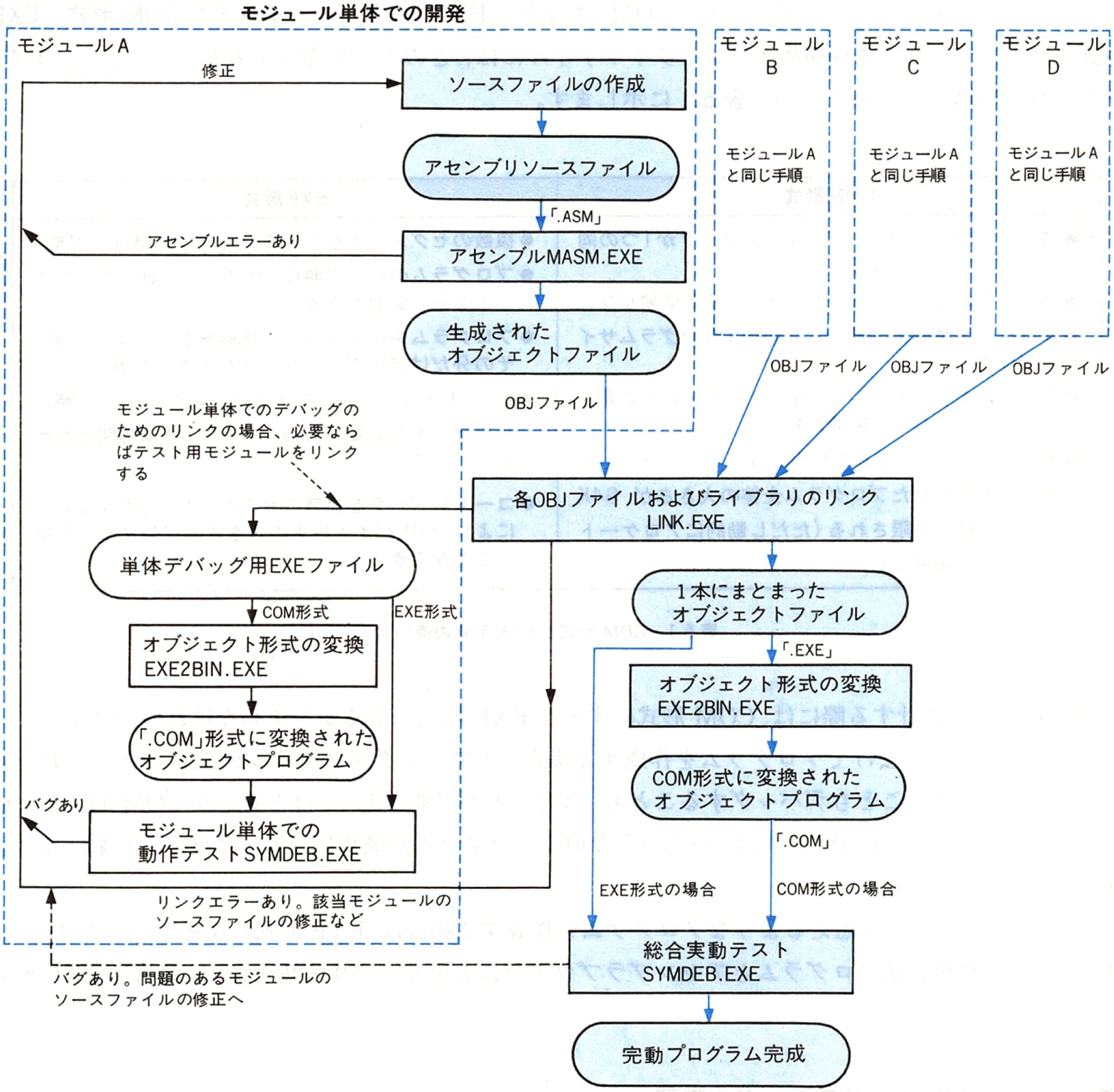

モジュール単体での開発
モジュールA
修正
ソースファイルの作成
アセンブリソースファイル
「.ASM」
アセンブルエラーあり
アセンブルMASM.EXE
生成された
オブジェクトファイル
OBJファイル
モジュールB
モジュールAと同じ手順
モジュールC
モジュールAと同じ手順
モジュールD
モジュールAと同じ手順
OBJファイル
OBJファイル
OBJファイル
モジュール単体でのデバッグのためのリンクの場合、必要ならばテスト用モジュールをリンクする
各OBJファイルおよびライブラリのリンク
LINK.EXE
単体デバッグ用EXEファイル
COM形式
EXE形式
オブジェクト形式の変換
EXE2BIN.EXE
「.COM」形式に変換された
オブジェクトプログラム
バグあり
モジュール単体での
動作テストSYMDEB.EXE
リンクエラーあり。該当モジュールのソースファイルの修正など
1本にまとまった
オブジェクトファイル
「.EXE」
オブジェクト形式の変換
EXE2BIN.EXE
COM形式に変換された
オブジェクトプログラム
「.COM」
EXE形式の場合
COM形式の場合
総合実動テスト
SYMDEB.EXE
バグあり。問題のあるモジュールのソースファイルの修正へ
完動プログラム完成

図 5.12　アセンブラによるソフトウェア開発全体の流れ

■ COM形式とEXE形式

すでに何度か触れていますが、MS-DOSの実行可能オブジェクトファイルの形式には2種類あります。ファイル名のファイルタイプが「.COM」および「.EXE」の2つで、それぞれ**COM形式**、**EXE形式**と呼びます。この2種類のオブジェクトファイルには、どのような違いがあるのでしょう。まず、それぞれの特徴をまとめたものを**表5.1**に示します。

COM形式	EXE形式
● コード(プログラム)、データ、スタックが1つの同じセグメントから成る ● プログラムのロード時にリロケートする必要がない ● リロケート情報を必要としないのでプログラムサイズが小さい ● ファイルが小さく、ロード時のリロケートを必要としないので、ロード時間が短い ● セグメントが1つであるため、コード、データ、スタックを合わせたプログラム全体の大きさが、64Kバイト以内に制限される(ただし動的にアロケートするデータ領域は別)	● 複数のセグメントから成るプログラム構成が可能 ● プログラムのロード時に、セグメント単位でリロケートを行う必要がある ● プログラム中にリロケート情報を含んでいるため、その分だけプログラムサイズが大きくなる ● COMモデルに比べてファイルが大きく、ロード時にリロケートが必要なため、ロードに多少時間がかかる ● コードやデータを複数のセグメントに分割することにより、64Kバイト以上の大きさのプログラムを扱うことができる

表5.1　COM形式とEXE形式の違い

プログラムを設計する際には、COM形式にするかEXE形式にするかを決めなければなりません。しかし、アセンブラだけでプログラムを作成する場合、オブジェクトファイルが64Kバイトを超えるようなものは、書くこともデバッグすることも、ほとんど不可能に近く、ほとんどがCOM形式を作成することになります。COM形式は、いろいろな面でプログラムが簡単になり、デバッグも楽になります。

しかし、64Kバイトを超えるようなプログラムを作成する場合は、EXE形式にするしかありません。また、ほかの高級言語プログラムとアセンブラプログラムとをリンクする場合にも、通常はEXE形式になります。

■ モジュール別プログラミングの実習

次節からは、モジュール別プログラミングにより、実際にプログラムを開発しながら、その過程を見ていきます。例題としては、四則演算のできる16進の電卓プログラムを作成してみましょう。この電卓の操作は、**図5.13**に示すように普通の電卓と同じですが、数値は16進数ですので0～9、A～Fのキーを、また演算子の「×」「÷」にはそれぞれ「*」「/」のキーを使います。

A>CALC⏎……………16進電卓のプログラムを起動する（プログラムファイル名は「CALC.COM」）

……………プログラムが起動すると入力OKのプロンプト「＃」が表示される

＃78/2＊3＝00B4……………$\frac{78_H}{2_H}\times 3_H$の計算。「＝」を入力した時点で答が表示される。答は$B4_H$
（2章のクラスタ番号からFATエントリ位置を求めた計算例）

＃78－2＋B＝0081……………$78_H-2_H+B_H$の計算。答は81_H（クラスタ番号から論理セクタ番号を求める計算例）

＃^C……………………………………Ctrl-Cの入力で、このプログラムを終了する

A>

この電卓は、16進数4桁の四則演算ができる。普通の電卓と同様に、演算子による優先順位はなく、入力された順に計算される。ただし、入力途中の結果は表示されない。

図 5.13　16 進電卓の操作例

16 進電卓プログラムの設計

16 進電卓プログラム全体の基本的な処理の流れは図 5.14 のようになります。

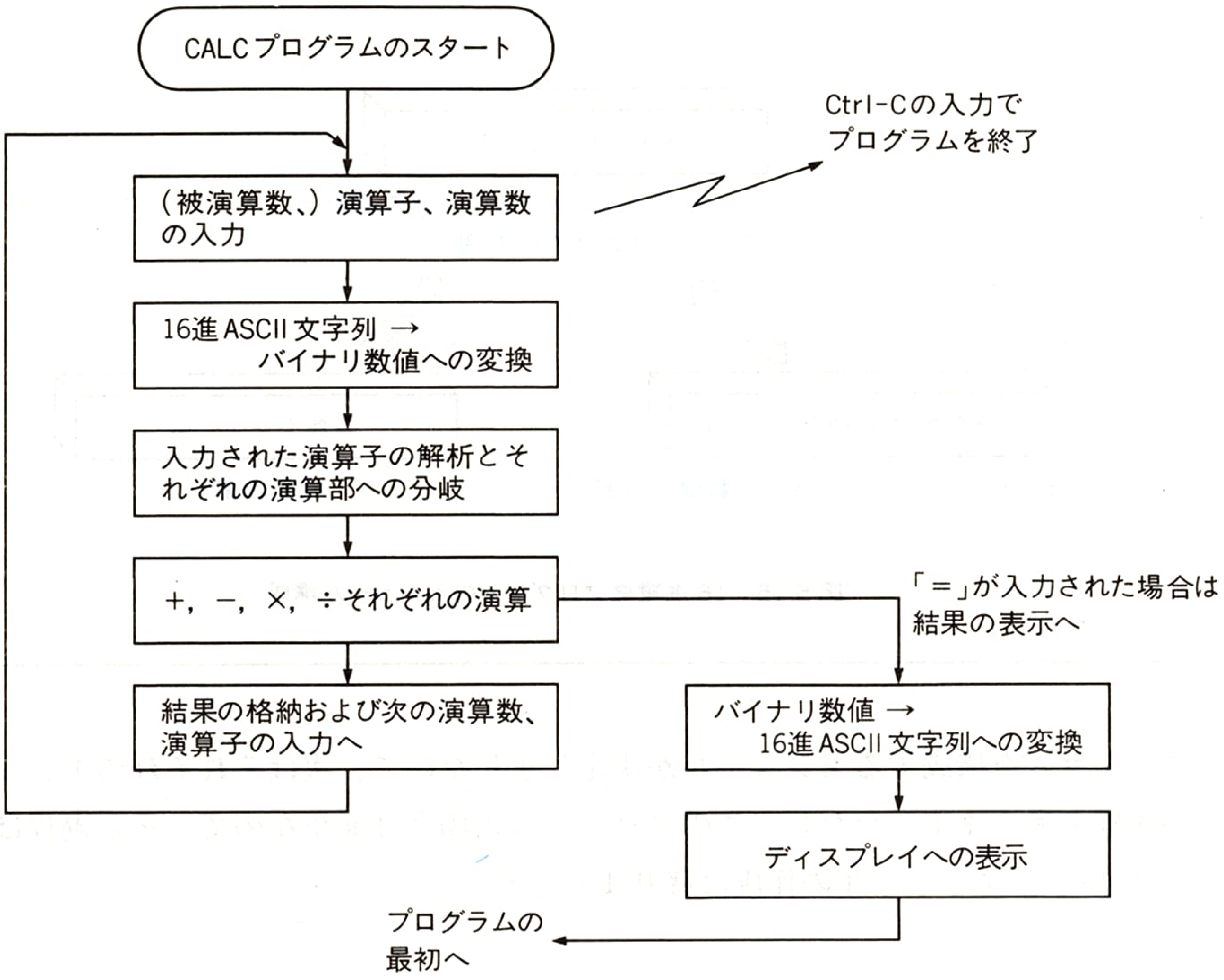

CALCプログラムのスタート
Ctrl-Cの入力で
プログラムを終了
（被演算数、）演算子、演算数の入力
16進ASCII文字列 → バイナリ数値への変換
入力された演算子の解析とそれぞれの演算部への分岐
＋，－，×，÷それぞれの演算
「＝」が入力された場合は
結果の表示へ
結果の格納および次の演算数、演算子の入力へ
バイナリ数値 → 16進ASCII文字列への変換
ディスプレイへの表示
プログラムの
最初へ

図 5.14　16 進電卓プログラムの基本処理の流れ

ここで、ソフトウェア開発の進め方の例を示しましょう。まず、各モジュールの枠組みを作り、そこにそれぞれの処理を当てはめ、枠の中身をプログラミングしていきます。いくつかのモジュールの中身が未完成でも、そこでの処理をとりあえず簡単なもので代用しておけば、プログラム全体の流れをテストすることができます。たとえば、文字列から数値への変換ルーチンが未完成でも、ある決まった数値を返すルーチンを作り、それで代用しておけば、とりあえずはプログラムを動かすことはできるのです。実際は代用ルーチンでも、正規の仕様どおりのルーチンを作ったつもりで、まず全体のプログラムを書いてしまうのです。このような設計の方法を**トップダウン法**と呼びます。これに対し、必要と思われるルーチンやモジュールをまず作り、それを組み合わせてプログラムを作っていく方法を**ボトムアップ法**と呼びます。

MS-DOSのアセンブラでは、どちらの方法でもプログラミングは可能です。どちらも一長一短はありますが、トップダウンの方が全体の見通しを立てやすく、思考形態としてはより高次元であると思われます。

では次に、この16進電卓プログラムのモジュール構成を考えてみましょう。このプログラムには、図5.14に挙げた処理が必要ですので、モジュール分けをすると、**図5.15**のようになるでしょう。

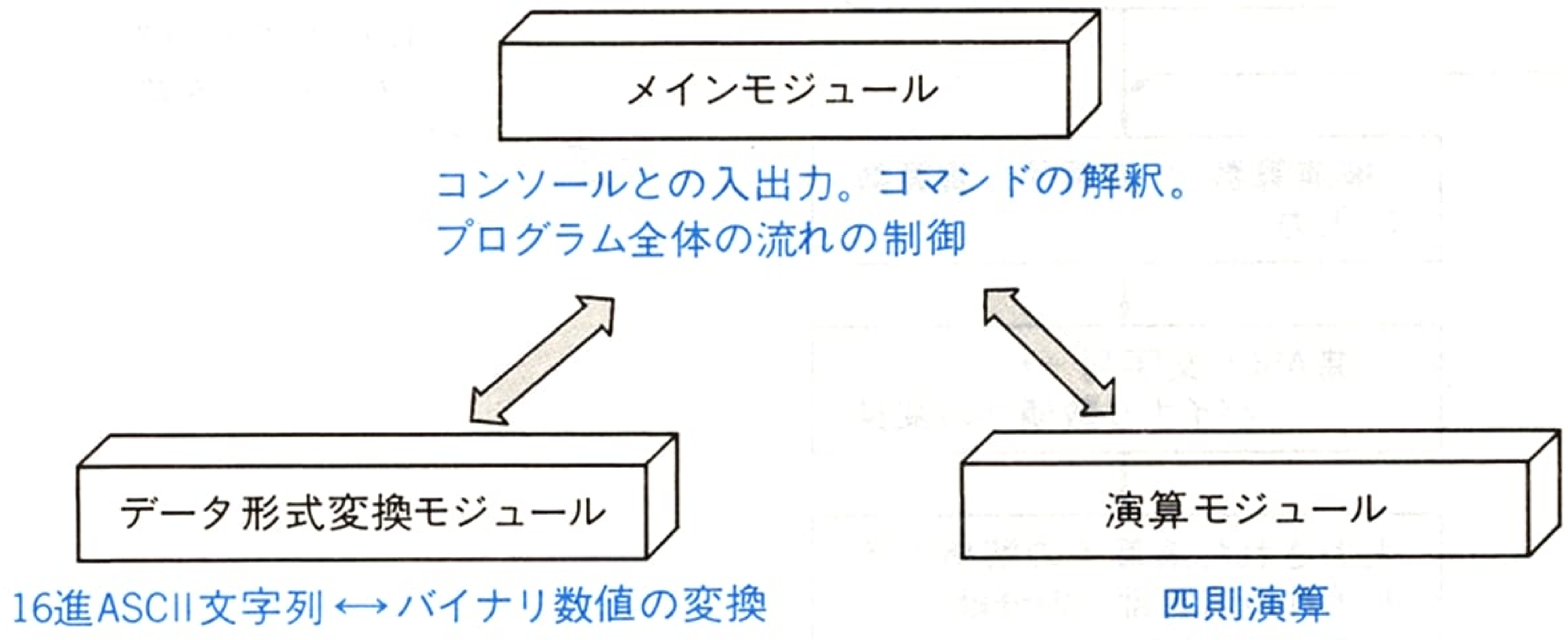

メインモジュール
コンソールとの入出力。コマンドの解釈。
プログラム全体の流れの制御
データ形式変換モジュール
16進ASCII文字列←→バイナリ数値の変換
演算モジュール
四則演算

図5.15　16進電卓プログラムのモジュール構成

さて、プログラムを構成するモジュールが決定しましたので、次はそれぞれのモジュールの入出力条件などの仕様を考えます。ただし、このプログラムの場合は簡単なので、その過程は省略して、直ちにアセンブリ・ソースファイルの作成に移りましょう。

5.3 アセンブリ・ソースファイルの書き方

MS-DOSの標準アセンブラMASMは、従来のマイクロコンピュータ用OSに付属していたような、機能の低いアセンブラとは違い、非常に高度な機能を持った強力なアセンブラです。

その第1は、リロケータブルアセンブラであることです。これはモジュール別プログラミングを可能にします。

第2は、マクロ機能を持つマクロアセンブラであることです。CPUの機能を擬似的に拡張したり、ソースファイルを短く効率的に、またわかりやすく書くことができます。

第3は、構造化機能を持っていることです。モジュール別プログラミングの機能とともに使うことによってプログラムを構造化して扱うことができ、トップダウンのプログラミングを可能にします。またデータの構造化も可能です。

このような高度な機能を持ったMASMを、どのように使いこなしていけばよいのでしょう。本節ではまずその基礎知識として、MASMのソースファイルを書く際の、プログラムのモジュール化、および構造化をサポートする機能について解説しましょう。

5.3.1 擬似命令の使い方

具体的な解説にはいる前に、8086系CPUのセグメント方式によるメモリ管理をサポートする擬似命令(CPU命令ではなく、アセンブラの動作を制御する命令)について、その基本的なものを説明しておきましょう。詳しくは8086系CPUやアセンブラの専門書を参照する必要がありますが、どのようなプログラムを書くにも、最低限、以下に挙げるような命令群を書かなければなりません。必要な命令は、モジュール別プログラミング法をとるかとらないか、COM形式かEXE形式かによって異なりますので、これについても簡単に触れておきましょう。

なお、次に挙げる擬似命令の実際のプログラミング例は、16進電卓プログラムのリスト(リスト5.1～5.3)にもあり、解説を加えていますので、随時参照してください。

■ SEGMENT

MS-DOSは(8086系CPUを使うコンピュータはすべて)、セグメント経由でメモリをアクセスするため、コード(プログラム)およびデータは、必ずいずれかのセグメントに属していなければなりません。その属するセグメントの始まりと終わりを任意の名前で指定するのが**SEGMENT擬似命令**です(図5.17参照)。

```
〈名前〉        SEGMENT        〈属性〉
                  ∫
〈名前〉        ENDS
```

SEGMENTからENDSの間のコードおよびデータは、その名前のセグメントに属します。この命令は、どんなコードやデータに関しても必ず必要で、それらのコードやデータは、この2つの命令で囲まれた間に書かなければなりません。

SEGMENT擬似命令の〈属性〉に関しては、とりあえずは必要ありません。とくにCOM形式の場合は、まったく書く必要はありません。EXE形式の場合は、STACK属性を持ったセグメントが最低1つは必要になります。COM形式の場合は、セグメントは1つ(すべてのセグメントが同じ名前)でなければなりませんが、EXE形式の場合はセグメントをいくつ持ってもかまいません。

■ STACK

COM形式の場合は、MS-DOSシステムがスタックを自動的に設定してくれます。この点からも、COM形式はプログラミングが楽であり、簡単なプログラムに向いています。

EXE形式の場合は、スタック領域をユーザープログラムで用意しなければなりません。セグメントをスタックセグメントに割り当てるには、次のように記述します(図5.19、図5.20参照)。

```
〈名前〉      SEGMENT   STACK
              DB        〈スタックサイズ〉   DUP   (?)
〈名前〉      ENDS
```

EXE形式では、このようにして割り当てたスタックセグメントが最低1つは必要です。STACK属性を持つセグメントは、リンク時にスタック領域として認識され、ロード実行時には、MS-DOSシステムによって、スタックを指すレジスタが、スタック領域として割り当てたこのセグメントを指すように設定されます。

■ ASSUME

セグメントレジスタが、どのセグメントベースを指しているかによって、オフセットの値が異なります。場合によってはアクセスできないかもしれません。このため、セグメントレジスタがどのセグメントベースを指しているかを指示するのが**ASSUME擬似命令**です。つまり使用する(参照する)セグメントに、さきのSEGMENT擬似命令で設定したセグメント名を対応付ける擬似命令です(図5.17参照)。

```
ASSUME     〈セグメントレジスタ名〉: 〈セグメント名〉
```

COM形式の場合は、セグメントは1つであり、すべてのセグメントレジスタは同じ値なので、このASSUME擬似命令を1度だけ書いておけばよいことになります。EXE形式の場合は、参照するセグメントレジスタのセグメントベースを変更するたびに、このASSUME擬似命令で指示する必要があります。

■ ORG

アセンブラがコードを生成するための最初のロケーション(アドレス)を与える擬似命令です。COM形式のプログラムは、オフセット0100Hから実行され、それ以前の0000H～00FFHの部分は、PSPとしてMS-DOSシステムによって予約されています。そのため、プログラムのコードが0100Hから生成されるように、**ORG擬似命令**でプログラムの開始アドレスを指定しなければなりません。

COM形式のメインモジュールには、このORG擬似命令を次のように書き、その直後にラベルを付けます(図5.17、図5.18参照)。

```
            ORG   100H
    START：
```

このラベル(ここでは「START：」)がCOM形式での実行開始アドレスとなります。

サブモジュールでは、ORGを指定する必要はありません。またEXE形式の場合も必要ありません(図5.19、図5.20参照)。

■ END

プログラムの終わりを示す擬似命令です(図5.17参照)。

```
    END  [<スタートアドレス>]
```

EXE形式のメインモジュールでは、プログラムの開始アドレスを指定します。COM形式では、「ORG　100H」の直後の、オフセット0100Hをアドレスとするラベル名を書かなければなりません。COM形式では、必ずオフセット0100Hから実行されるので、これは当然です。

EXE形式では、どのアドレスでもかまわず、指定したシンボルのアドレスから実行が開始されます。指定しなかった場合は最初にリンクしたモジュールのオフセット0000Hから実行されますが、これはなるべく積極的に指定する方がよいでしょう。

以上のアセンブラの命令は、プログラムをモジュールに分けずに、単体で作成する場合でも、必ず必要な最低限のものです。

次に、プログラムのモジュール化、構造化をサポートする命令の基本的なものについて解説しましょう。

■ PROC

PROCはプロシージャ(手続き)のことで、サブルーチンの始めと終わりを宣言し、その構造を明確にします。PROC宣言時のオプションにより、そのサブルーチンをコールする際のNEAR／FARを指定しておくと、コールから戻る際のリターン命令のNEAR／FARが自動的に決定されるので、プログラミングが楽になります(図5.19、20参照)。

〈手続き名〉　PROC　［NEAR］　または　［FAR］
〜
〈手続き名〉　ENDP

■ PUBLIC、EXTRN

プログラムをいくつかのモジュールに分割する場合には、ほかのモジュールの手続きやデータを互いに参照するために、「このモジュールでは、ほかのモジュール内の、これと、これと、…、これのルーチンやデータを使いますよ」とか、「このモジュールでは、ほかのモジュールからの、これと、これと、…、これのルーチンやデータの参照を許可しますよ」などの宣言をしなければなりません。

モジュール内で使われるシンボル(数、変数、PROCラベルなどのラベル)は、そのままではそのシンボルが含まれるモジュールの中でしか通用しませんが、PUBLIC宣言したシンボルはほかのモジュールに対して公(おおやけ)になり、ほかのモジュール内でも使うことができます(図5.18、図5.20参照)。

このことを逆に見れば、PUBLIC宣言をしていないシンボルは、ほかのモジュールで使っているものと同じ名前を使うことが可能です。つまりそれらの各シンボルは、そのモジュール内でしか有効ではないので、重複してもかまわないのです。

PUBLIC　〈シンボル〉

ほかのモジュールでPUBLIC宣言されているシンボルを、自分のモジュールで参照するには、そのシンボルをEXTRN宣言しておく必要があります。つまり、「ほかのモジュールのシンボルを使いますよ」という宣言です。(図5.18、図5.20参照)。

EXTRN　〈シンボル〉：〈型〉

EXTRN宣言したシンボルは、自分のモジュール内で定義されている通常のシンボルと同じように扱うことができます。ただしそのシンボルは、アセンブル時にはアドレスが決まらないため保留された値となっており、リンクのときに決定されます(図5.4および図5.16参照)。

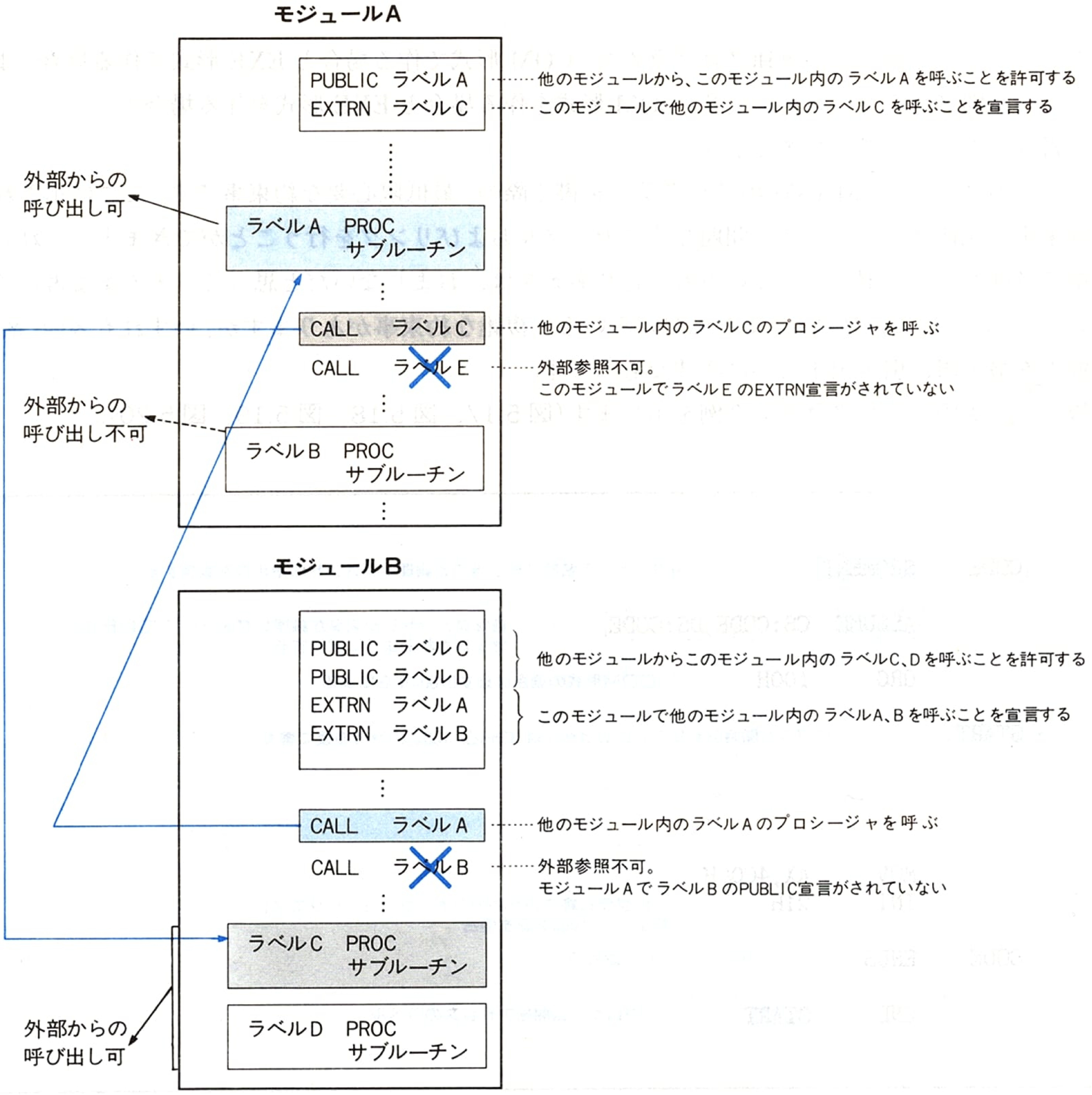

モジュールA
PUBLIC　ラベルA
EXTRN　ラベルC
他のモジュールから、このモジュール内のラベルAを呼ぶことを許可する
このモジュールで他のモジュール内のラベルCを呼ぶことを宣言する
外部からの呼び出し可
ラベルA　PROC
サブルーチン
CALL　ラベルC
他のモジュール内のラベルCのプロシージャを呼ぶ
CALL　ラベルE
外部参照不可。
このモジュールでラベルEのEXTRN宣言がされていない
外部からの呼び出し不可
ラベルB　PROC
サブルーチン
モジュールB
PUBLIC　ラベルC
PUBLIC　ラベルD
EXTRN　ラベルA
EXTRN　ラベルB
他のモジュールからこのモジュール内のラベルC、Dを呼ぶことを許可する
このモジュールで他のモジュール内のラベルA、Bを呼ぶことを宣言する
CALL　ラベルA
他のモジュール内のラベルAのプロシージャを呼ぶ
CALL　ラベルB
外部参照不可。
モジュールAでラベルBのPUBLIC宣言がされていない
ラベルC　PROC
サブルーチン
外部からの呼び出し可
ラベルD　PROC
サブルーチン

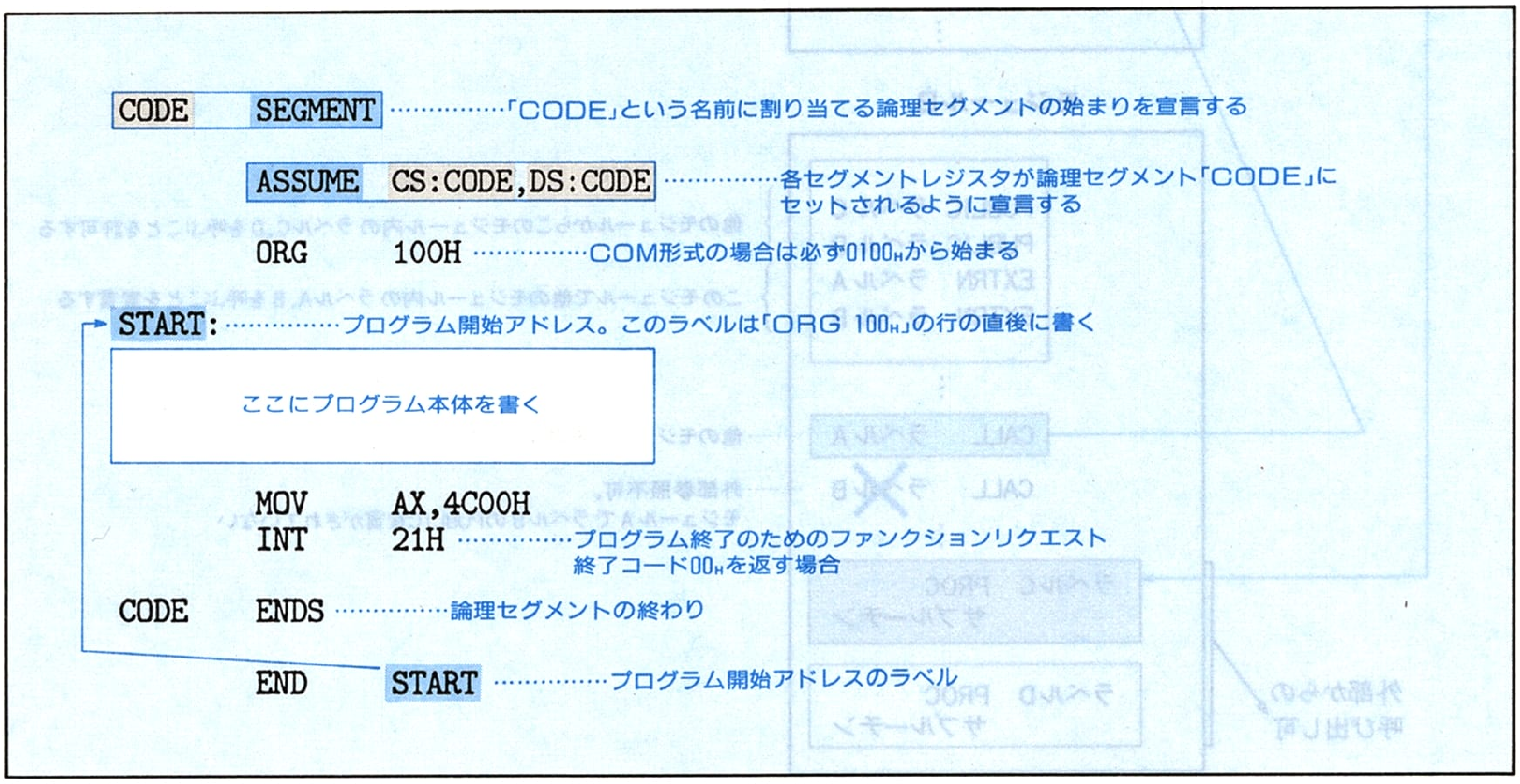
図 5.16　PUBLIC および EXTRN の働き

5.3.2　オブジェクト形式別ソースファイルの書き方

モジュールに分割しない単独プログラムを、COM形式で作る場合とEXE形式で作る場合、また、モジュール別プログラミングにより、COM形式を作る場合とEXE形式を作る場合の、ソースファイルの書き方をまとめて示しましょう。

ここにあげるのは、MASMのプログラムを書く際の、最低限必要な約束事です。ここに示されている約束事が記述されていれば、問題なくアセンブルおよびリンクを行うことができます。これらの約束事の意味をまだ理解していない方も、とりあえずは、おまじないだと思って、そのまま書いてください。ここにあげる以外にも、まだまだいろいろと複雑な約束事がありますが、いずれも8086系CPUの能力を最大限に引き出すために必要なものです。

以下に、実際のプログラミング例を示します(図5.17、図5.18、図5.19、図5.20)。



```
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE
        ORG     100H
START:
        ここにプログラム本体を書く
        MOV     AX,4C00H
        INT     21H
CODE    ENDS
        END     START
```

CODE SEGMENT ……「CODE」という名前に割り当てる論理セグメントの始まりを宣言する

ASSUME CS:CODE,DS:CODE ……各セグメントレジスタが論理セグメント「CODE」にセットされるように宣言する

ORG 100H ……COM形式の場合は必ず0100_Hから始まる

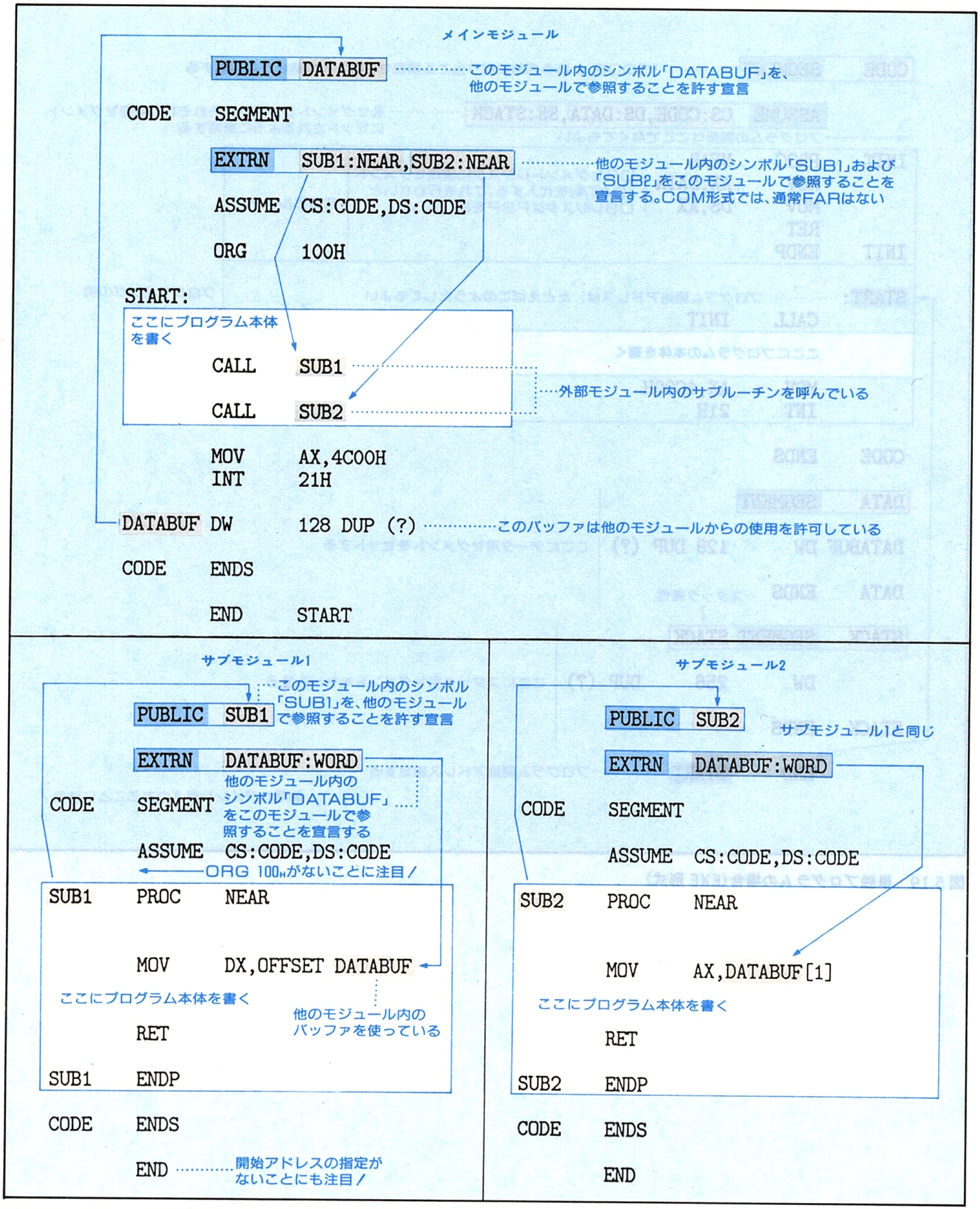
START: ……プログラム開始アドレス。このラベルは「ORG 100_H」の行の直後に書く

INT 21H ……プログラム終了のためのファンクションリクエスト 終了コード00_Hを返す場合

CODE ENDS ……論理セグメントの終わり

END START ……プログラム開始アドレスのラベル

図5.17　単独プログラムの場合(COM形式)



メインモジュール

```
        PUBLIC  DATABUF
CODE    SEGMENT
        EXTRN   SUB1:NEAR,SUB2:NEAR
        ASSUME  CS:CODE,DS:CODE
        ORG     100H
START:
        CALL    SUB1
        CALL    SUB2
        MOV     AX,4C00H
        INT     21H
DATABUF DW      128 DUP (?)
CODE    ENDS
        END     START
```

このモジュール内のシンボル「DATABUF」を、他のモジュールで参照することを許す宣言

他のモジュール内のシンボル「SUB1」および「SUB2」をこのモジュールで参照することを宣言する。COM形式では、通常FARはない

ここにプログラム本体を書く

外部モジュール内のサブルーチンを呼んでいる

このバッファは他のモジュールからの使用を許可している

サブモジュール1

```
        PUBLIC  SUB1
        EXTRN   DATABUF:WORD
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE
SUB1    PROC    NEAR
        MOV     DX,OFFSET DATABUF
        RET
SUB1    ENDP
CODE    ENDS
        END
```

このモジュール内のシンボル「SUB1」を、他のモジュールで参照することを許す宣言

他のモジュール内のシンボル「DATABUF」をこのモジュールで参照することを宣言する

ORG 100Hがないことに注目！

ここにプログラム本体を書く

他のモジュール内のバッファを使っている

開始アドレスの指定がないことにも注目！

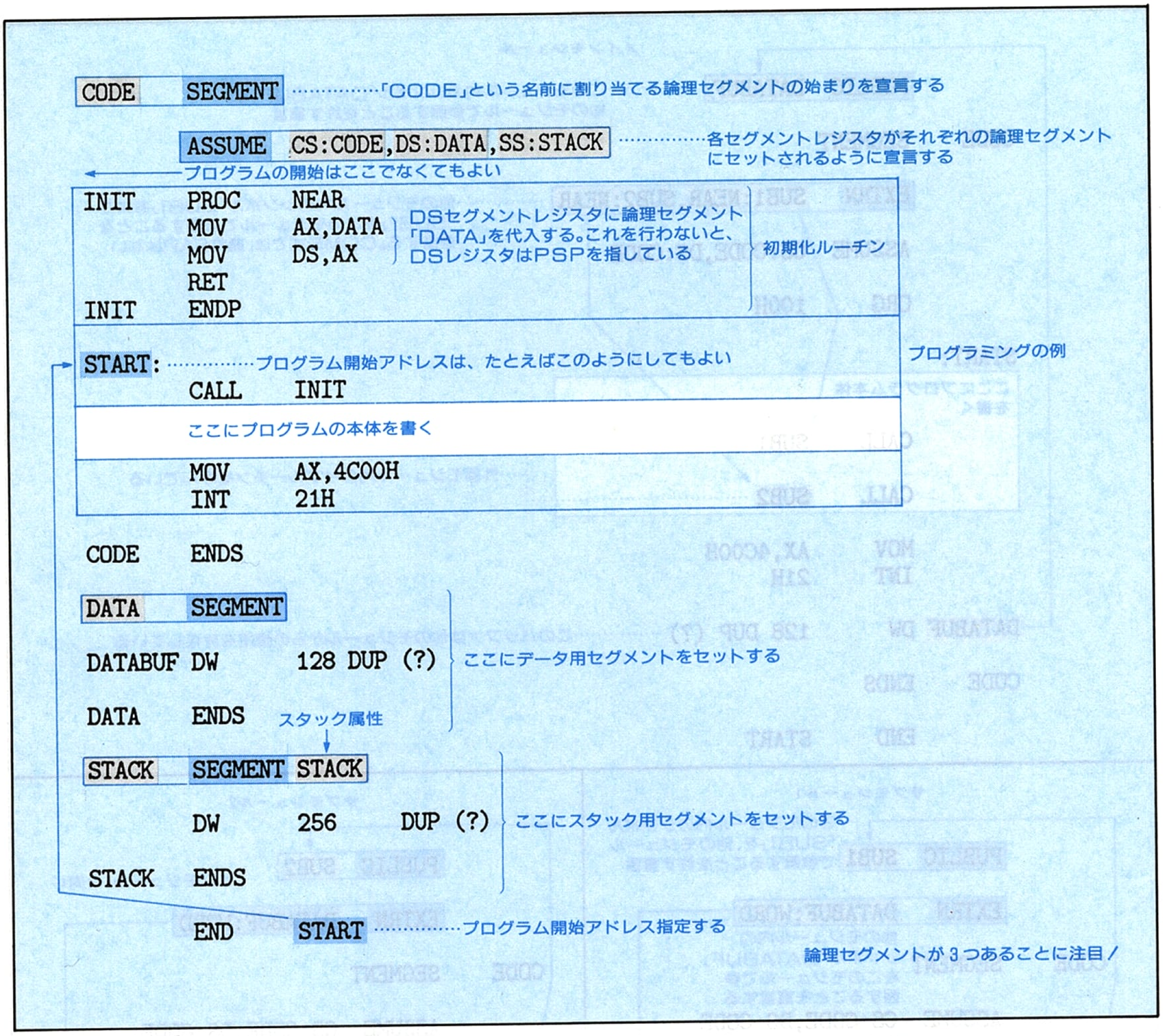
サブモジュール2

```
        PUBLIC  SUB2
        EXTRN   DATABUF:WORD
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE
SUB2    PROC    NEAR
        MOV     AX,DATABUF[1]
        RET
SUB2    ENDP
CODE    ENDS
        END
```

サブモジュール1と同じ

ここにプログラム本体を書く

図 5.18　モジュール別プログラミングによる場合（COM 形式）



```
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA,SS:STACK
INIT    PROC    NEAR
        MOV     AX,DATA
        MOV     DS,AX
        RET
INIT    ENDP
START:
        CALL    INIT
        MOV     AX,4C00H
        INT     21H
CODE    ENDS
DATA    SEGMENT
DATABUF DW      128 DUP (?)
DATA    ENDS
STACK   SEGMENT STACK
        DW      256     DUP (?)
STACK   ENDS
        END     START
```

「CODE」という名前に割り当てる論理セグメントの始まりを宣言する

各セグメントレジスタがそれぞれの論理セグメントにセットされるように宣言する

プログラムの開始はここでなくてもよい

DSセグメントレジスタに論理セグメント「DATA」を代入する。これを行わないと、DSレジスタはPSPを指している

初期化ルーチン

プログラム開始アドレスは、たとえばこのようにしてもよい

プログラミングの例

ここにプログラムの本体を書く

ここにデータ用セグメントをセットする

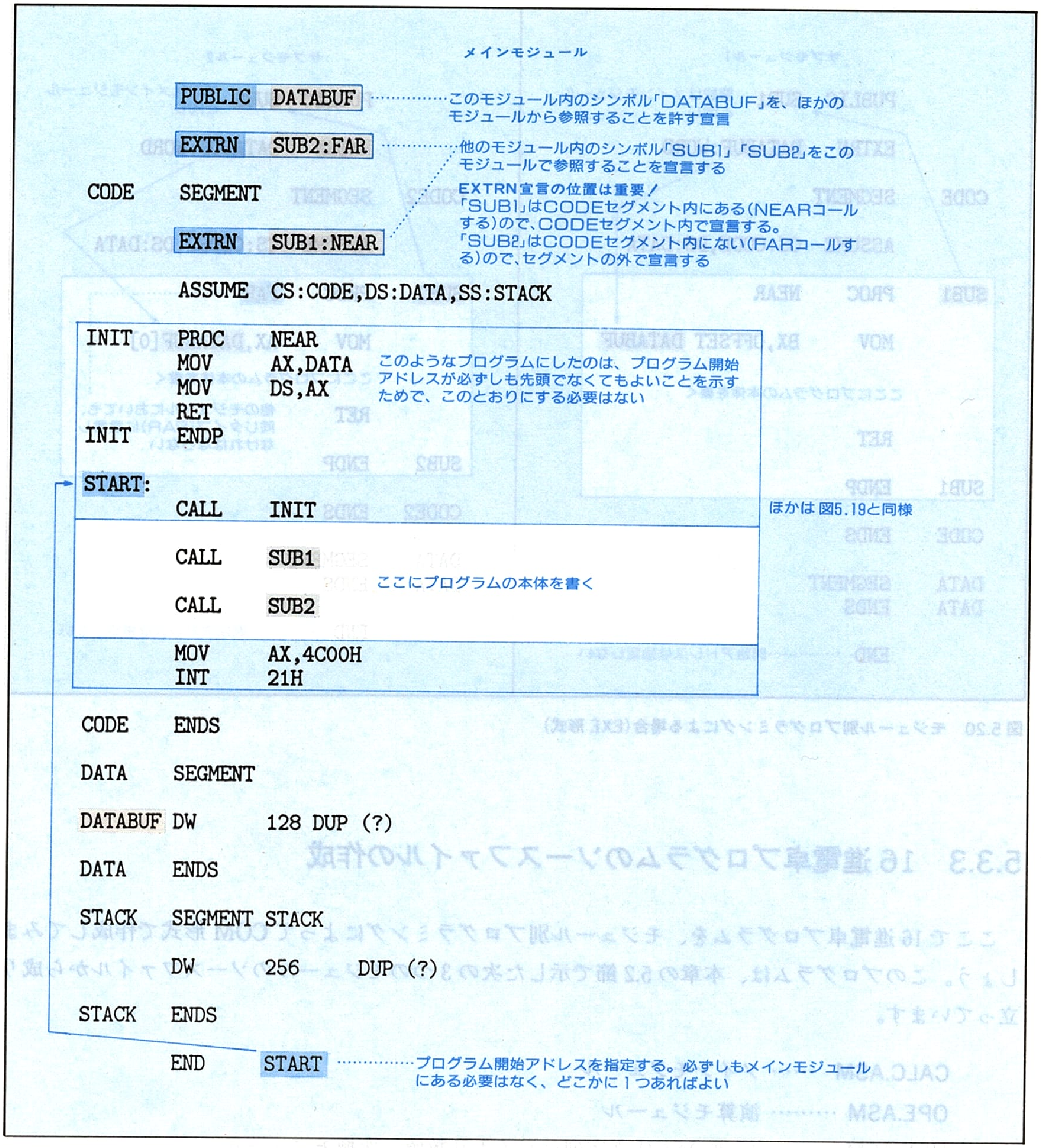
スタック属性

ここにスタック用セグメントをセットする

プログラム開始アドレス指定する

論理セグメントが3つあることに注目！

図 5.19　単独プログラムの場合（EXE 形式）



```
        PUBLIC  DATABUF
        EXTRN   SUB2:FAR
CODE    SEGMENT
        EXTRN   SUB1:NEAR
        ASSUME  CS:CODE,DS:DATA,SS:STACK
INIT    PROC    NEAR
        MOV     AX,DATA
        MOV     DS,AX
        RET
INIT    ENDP
START:
        CALL    INIT
        CALL    SUB1
        CALL    SUB2
        MOV     AX,4C00H
        INT     21H
CODE    ENDS
DATA    SEGMENT
DATABUF DW      128 DUP (?)
DATA    ENDS
STACK   SEGMENT STACK
        DW      256     DUP (?)
STACK   ENDS
        END     START
```

このモジュール内のシンボル「DATABUF」を、ほかのモジュールから参照することを許す宣言

他のモジュール内のシンボル「SUB1」「SUB2」をこのモジュールで参照することを宣言する

EXTRN宣言の位置は重要！
「SUB1」はCODEセグメント内にある(NEARコールする)ので、CODEセグメント内で宣言する。
「SUB2」はCODEセグメント内にない(FARコールする)ので、セグメントの外で宣言する

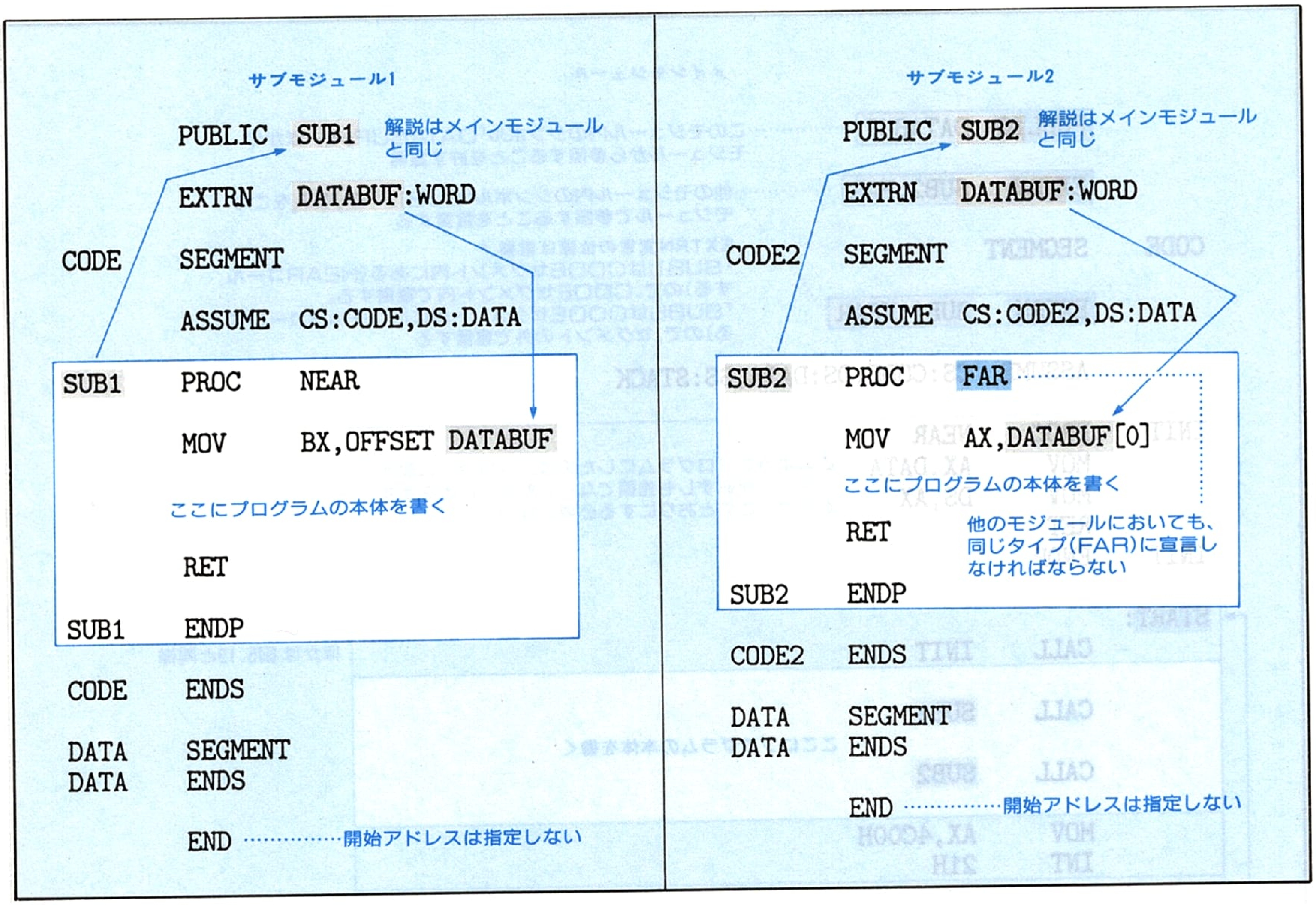
このようなプログラムにしたのは、プログラム開始アドレスが必ずしも先頭でなくてもよいことを示すためで、このとおりにする必要はない

ほかは 図5.19と同様

ここにプログラムの本体を書く

プログラム開始アドレスを指定する。必ずしもメインモジュールにある必要はなく、どこかに１つあればよい

— 図 5.20 — (次ページに続く)



サブモジュール1

```
         PUBLIC  SUB1
         EXTRN   DATABUF:WORD
CODE     SEGMENT
         ASSUME  CS:CODE,DS:DATA
SUB1     PROC    NEAR
         MOV     BX,OFFSET DATABUF

         RET
SUB1     ENDP
CODE     ENDS

DATA     SEGMENT
DATA     ENDS

         END
```

解説はメインモジュールと同じ

ここにプログラムの本体を書く

開始アドレスは指定しない

サブモジュール2

```
         PUBLIC  SUB2
         EXTRN   DATABUF:WORD
CODE2    SEGMENT
         ASSUME  CS:CODE2,DS:DATA
SUB2     PROC    FAR
         MOV     AX,DATABUF[0]

         RET
SUB2     ENDP
CODE2    ENDS

DATA     SEGMENT
DATA     ENDS

         END
```

解説はメインモジュールと同じ

ここにプログラムの本体を書く

他のモジュールにおいても、同じタイプ(FAR)に宣言しなければならない

開始アドレスは指定しない

図5.20　モジュール別プログラミングによる場合(EXE形式)

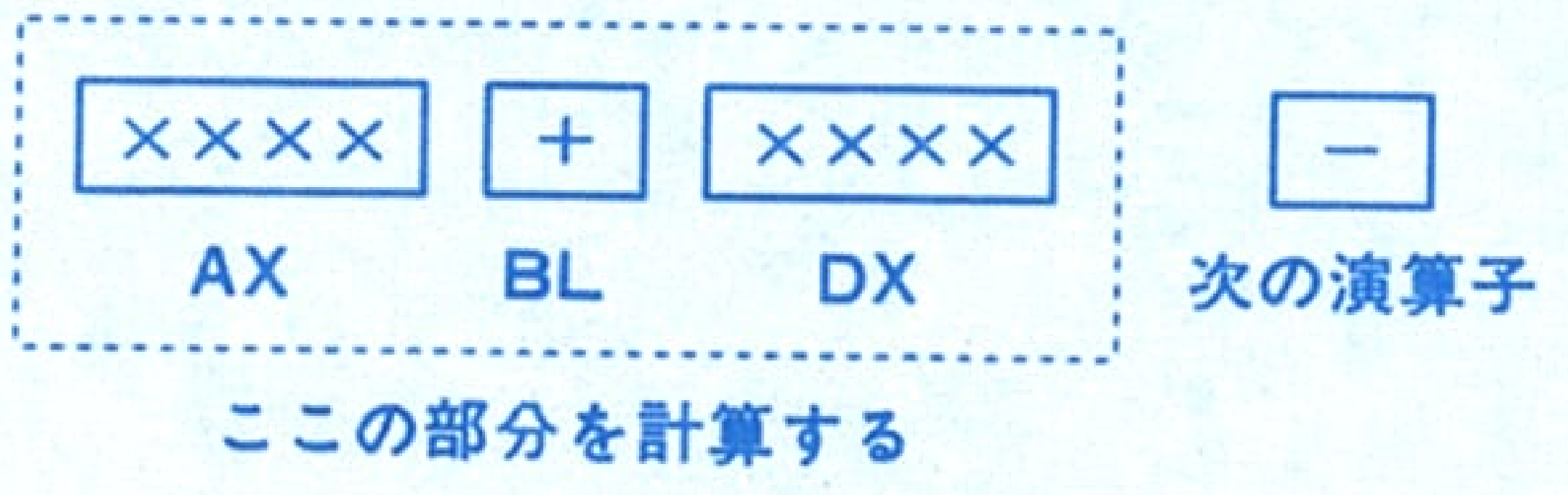
5.3.3　16進電卓プログラムのソースファイルの作成

ここで16進電卓プログラムを、モジュール別プログラミングによってCOM形式で作成してみましょう。このプログラムは、本章の5.2節で示した次の3つのモジュールのソースファイルから成り立っています。

CALC.ASM ……メインモジュール
OPE.ASM ………演算モジュール
HEX.ASM ………16進ASCII文字列↔バイナリ数値　変換モジュール

次に、これまで述べてきた各種の約束事に従って書かれた、これらのソースファイルを示します(リスト5.1、リスト5.2、リスト5.3)。なお、メインモジュールについてはその概念のフローチャートを示しておきます(図5.21)。

```
;; CALC.ASM       Hexadecimal Caluculator
;;                Main Program

CODE     SEGMENT  ……………論理セグメント「CODE」の始まり
         EXTRN    BINADD:NEAR,BINSUB:NEAR,BINMUL:NEAR,BINDIV:NEAR } 他のモジュールの
         EXTRN    HEXTOBIN:NEAR,BINTOHEX:NEAR                     } ラベルを使用する
                                                                    ことを宣言する
         ASSUME   CS:CODE,DS:CODE ……………セグメントレジスタが「CODE」にセットされるように宣言する
         ORG      100H ……………COM形式なので、プログラムの
                                開始は0100Hから

START:   XOR      AX,AX
         MOV      NUMBER,AX
         MOV      AL,’=’
         MOV      OPERATOR,AL
         MOV      AH,2
         MOV      DL,’#’
         INT      21H
         MOV      SI,OFFSET HEXSTR ……………16進文字列を格納するアドレス
         MOV      CX,4 …………… 4桁まで入力するためのカウンタをセットする
GETFSTNUM:
         CALL     GETNUM ……………16進文字列を入力するサブルーチンをコールする
         MOV      BL,AL
         CMP      CX,4       } 残りの桁数が4なら、つまり1文字も入力されて
         JE       GETFSTNUM  } いなければもう一度入力し直す
         CALL     ISOPE      } 入力を終了する原因になった文字が演算子かどうか
         JNE      GETFSTNUM  } 調べる。演算子でなければ入力を続ける
         JMP      SHORT CALC ……………最初の入力式の演算へ
GETNEXTNUM:
         MOV      SI,OFFSET HEXSTR } 次の4桁までの16進数を入力する
         MOV      CX,4             }
GETCONT:
         CALL     GETNUM ……………16進文字列を入力するサブルーチンをコールする
         MOV      BL,AL
         CALL     ISOPE    } 演算子かどうか。演算子でなければ入力を続ける
         JNE      GETCONT  }
         CMP      CX,4 ……………残りの桁数が4なら何も入力されていないので、もう一度入力する
         JE       GETCONT
CALC:
         MOV      DL,BL  } 演算子を画面に表示する
         CALL     PUTC   }
         MOV      [SI],BYTE PTR ’$’                          }
         MOV      SI,OFFSET HEXSTR                           } 16進文字列の末尾(SIレジスタが指している)
         CALL     HEXTOBIN ……………16進文字列をバイナリ数値へ } に’$’を入力し、バイナリ数値に変換する
                              変換するサブルーチン            }

         MOV      DX,NUMBER      } AXレジスタに被演算数、DXレジスタに演算数をセットする。
         XCHG     AX,DX          } 「NUMBER」には今までの値が、「OPERATOR」には1つ前
         XCHG     BL,OPERATOR    } の演算子がはいっているようにする。各演算子の処理へ分岐する
         CMP      BL,’+’         }
         JE       ADDITION       }
         CMP      BL,’-’         }
         JE       SUBTRACTION    }
         CMP      BL,’*’         }
```



— リスト 5.1 — (次ページ以下に続く)

```
        JE      MULTIPLICATION
        CMP     BL,’/’
        JE      DIVISION
        MOV     AX,DX
        JMP     SHORT STORENUM
ADDITION:
        CALL    BINADD          } 加算へ
        JMP     SHORT STORENUM  }
SUBTRACTION:
        CALL    BINSUB          } 減算へ
        JMP     SHORT STORENUM  }
MULTIPLICATION:
        CALL    BINMUL          } 乗算へ
        JMP     SHORT STORENUM  }
DIVISION:
        CALL    BINDIV ……………除算へ
STORENUM:
        MOV     NUMBER,AX ……………得られた数を格納する
        MOV     DL,OPERATOR     } 次の演算子が「＝」なら値(答)を表示する処理へ、
        CMP     DL,’=’          } そうでなければ次の値の入力へ
        JNE     GETNEXTNUM      }
        MOV     SI,OFFSET HEXSTR } 最終的な値(答)を16進文字列に変換する
        CALL    BINTOHEX         }
        MOV     AH,9             文字列出力のファンクションリクエスト。
        MOV     DX,OFFSET HEXSTR 答をディスプレイへ表示する
        INT     21H
        MOV     DL,0DH
        CALL    PUTC             次の入力のために改行する
        MOV     DL,0AH
        CALL    PUTC
        JMP     START

ISOPE   PROC
        CMP     AL,’=’
        JE      ISOPERET
        CMP     AL,’+’
        JE      ISOPERET
        CMP     AL,’-’           演算子かどうかをチェックするサブルーチン
        JE      ISOPERET           入力条件　AL→文字
        CMP     AL,’*’             出力条件　ゼロフラグ=「1」なら演算子
        JE      ISOPERET                           =「0」なら演算子ではない
        CMP     AL,’/’
ISOPERET:
        RET
ISOPE   ENDP
```

```
GETNUM   PROC
GETNNUM:
         CALL     GETC
         CMP      AL,'0'
         JL       NOTNUM
         CMP      AL,'9'
         JLE      SETNUM
         CMP      AL,'A'
         JL       NOTNUM
         CMP      AL,'F'
         JLE      SETNUM
         CMP      AL,'a'
         JL       NOTNUM
         CMP      AL,'f'
         JG       NOTNUM
SETNUM:
         JCXZ     GETNNUM
         MOV      [SI],AL
         INC      SI
         DEC      CX
         MOV      DL,AL
         CALL     PUTC
         JMP      SHORT GETNNUM
NOTNUM:
         RET
GETNUM   ENDP

GETC     PROC
         MOV      AH,8
         INT      21H
         RET
GETC     ENDP

PUTC     PROC
         MOV      AH,2
         INT      21H
         RET
PUTC     ENDP

NUMBER   DW       0
OPERATOR DB       '='
HEXSTR   DB       4 DUP (?),'$'

CODE     ENDS
         END      START
```

GETNUM PROC……16進数を表す文字列をキーボードから受け取るサブルーチン

入力条件 SI：格納するアドレス
入力条件 CX：最大桁数

出力条件 格納された文字列
出力条件 SI：次に格納する場所
出力条件 CX：残り桁数
出力条件 AL：入力を終了する原因になった文字

CALL GETC……キーボードから1文字入力する

'0'≦AL≦'9'かどうか

'A'≦AL≦'F'
'a'≦AL≦'f' かどうか

範囲外の文字が入力されたら戻る

JCXZ GETNNUM……CXレジスタが0なら最後の桁まで入力されているのでこれ以上入力できない。範囲外の文字が入力されるまで待つ

MOV [SI],AL / INC SI：文字を格納し、ポインタを1つ進める

DEC CX / MOV DL,AL / CALL PUTC：残り桁数のカウンタを－1して文字をディスプレイに表示する

JMP SHORT GETNNUM……次の文字の入力へ

GETNUM ENDP……サブルーチン「GETNUM」の終わり

MOV AH,8 / INT 21H：キーボードから1文字入力するサブルーチン
エコーバックしない

入力条件　なし
出力条件　AL：入力された文字の文字コード

MOV AH,2 / INT 21H：ディスプレイに1文字出力するサブルーチン

入力条件　DL：出力する文字
出力条件　なし

このように分離しておけば、たとえばCtrl-Cによる中断をしないように変更する場合にもここの変更だけですむ

NUMBER DW 0……中間結果の値を格納する

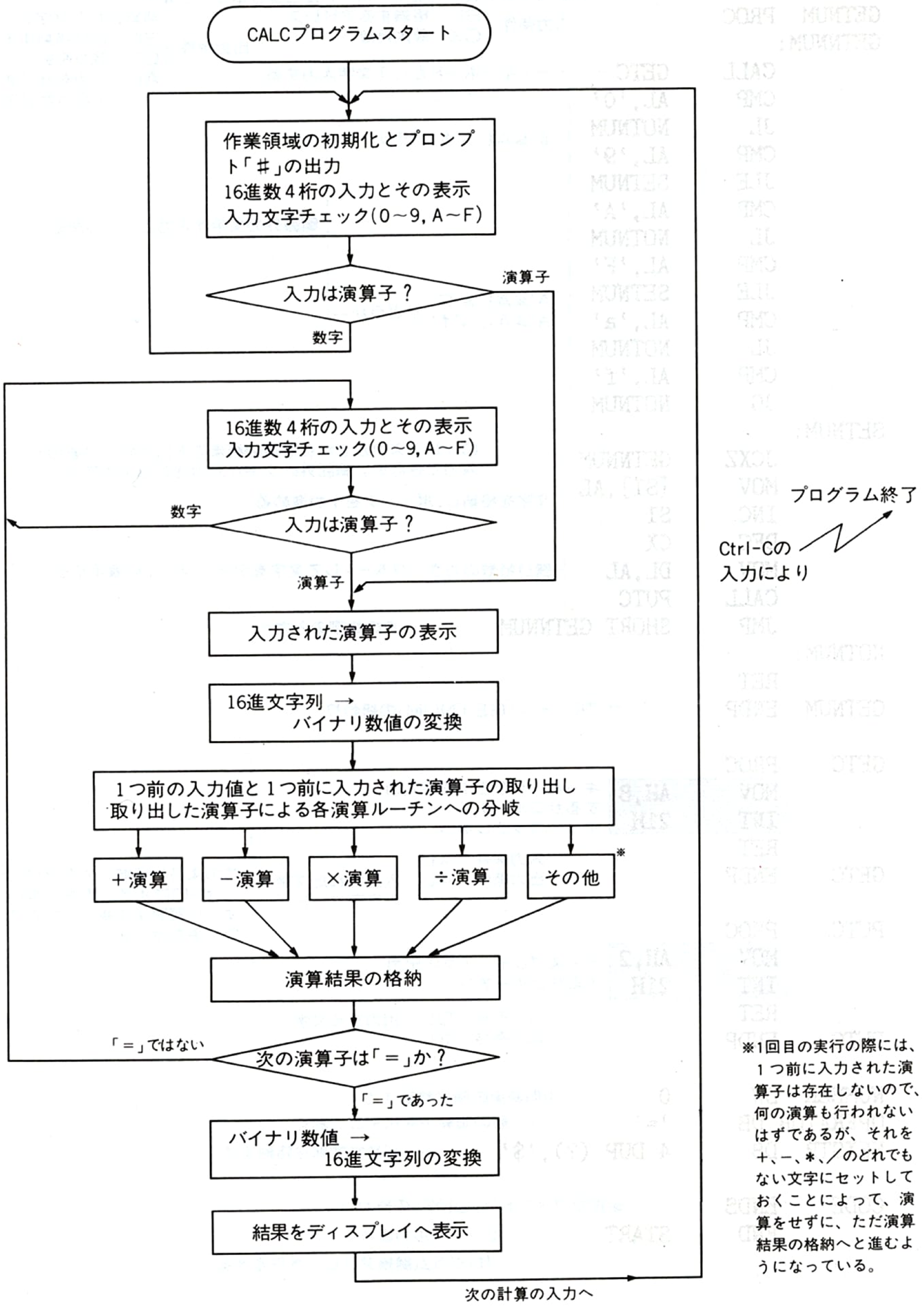
OPERATOR DB '='……1つ前の演算子を格納しておく

HEXSTR DB 4 DUP (?),'$'……16進文字列を格納する

CODE ENDS……論理セグメント「CODE」の終わり

END START……モジュールの終わり
プログラム開始アドレスを指定する

リスト 5.1　メインモジュール CALC.ASM


CALCプログラムスタート
作業領域の初期化とプロンプト「♯」の出力
16進数4桁の入力とその表示
入力文字チェック（0～9，A～F）
入力は演算子？
演算子
数字
16進数4桁の入力とその表示
入力文字チェック（0～9，A～F）
数字
入力は演算子？
演算子
入力された演算子の表示
16進文字列 → バイナリ数値の変換
1つ前の入力値と1つ前に入力された演算子の取り出し
取り出した演算子による各演算ルーチンへの分岐
＋演算
－演算
×演算
÷演算
その他
※
演算結果の格納
「＝」ではない
次の演算子は「＝」か？
「＝」であった
バイナリ数値 → 16進文字列の変換
結果をディスプレイへ表示
次の計算の入力へ
プログラム終了
Ctrl-Cの入力により

※1回目の実行の際には、1つ前に入力された演算子は存在しないので、何の演算も行われないはずであるが、それを＋、－、＊、／のどれでもない文字にセットしておくことによって、演算をせずに、ただ演算結果の格納へと進むようになっている。

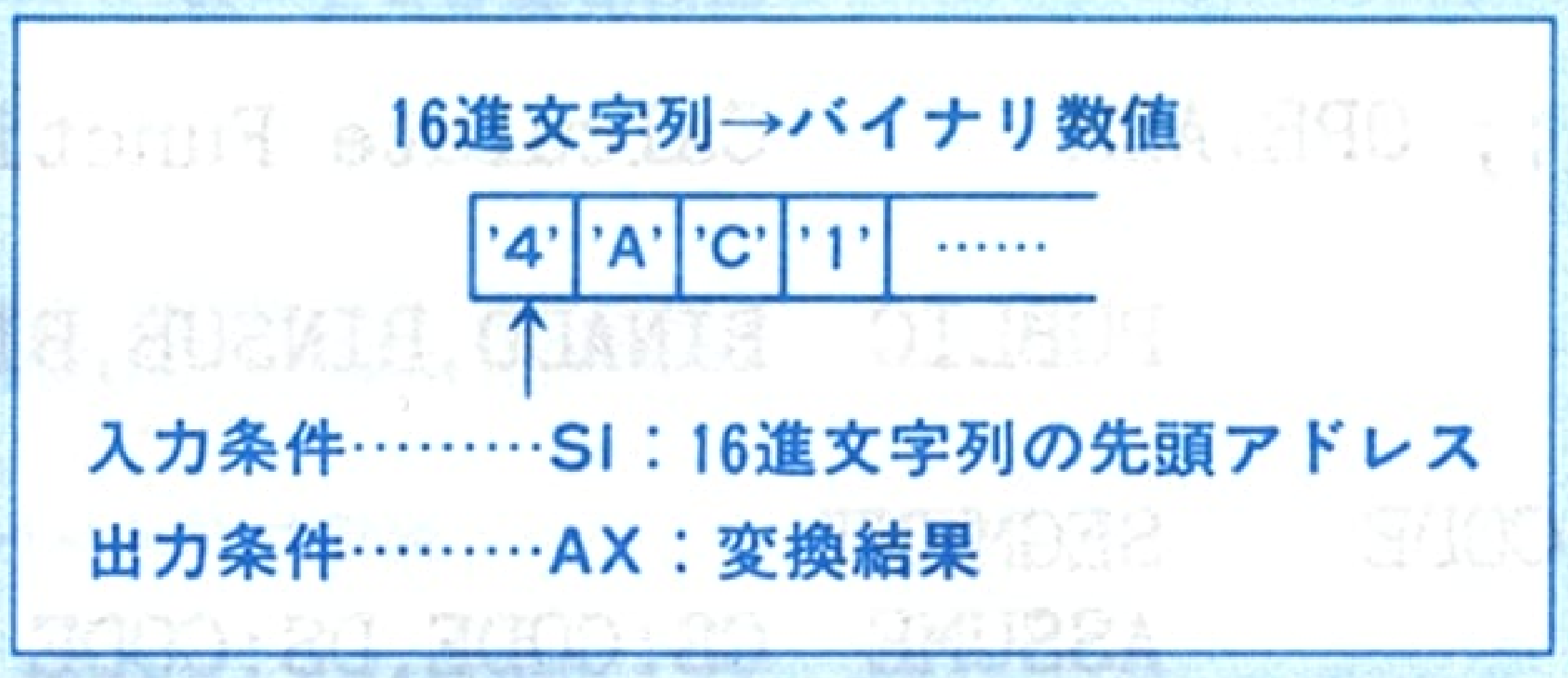
図 5.21　メインモジュール CALC.ASM の概略

```
                    四則演算モジュール
;; OPE.ASM          Calculate Functions

           PUBLIC   BINADD,BINSUB,BINMUL,BINDIV ……………この4つのサブルーチンを他のモジュールで
                                                          使用することを宣言する
CODE       SEGMENT ……………論理セグメント「CODE」の始まり
           ASSUME   CS:CODE,DS:CODE ……………セグメントレジスタが「CODE」にセットされるように宣言する

BINADD     PROC
           ADD      AX,DX
           RET                 加算のサブルーチン
BINADD     ENDP

BINSUB     PROC
           SUB      AX,DX
           RET                 減算のサブルーチン
BINSUB     ENDP                                     各ルーチンとも
                                                    入力条件 {AX：被演算数
                                                             {DX：演算数
BINMUL     PROC                                     出力条件  AX：結果
           MUL      DX
           RET                 乗算のサブルーチン
BINMUL     ENDP

BINDIV     PROC
           MOV      BX,DX
           XOR      DX,DX
           DIV      BX         除算のサブルーチン
           RET
BINDIV     ENDP

CODE       ENDS ……………論理セグメント「CODE」の終わり
           END ……………モジュールの終わり
```

リスト 5.2 演算モジュール OPE.ASM

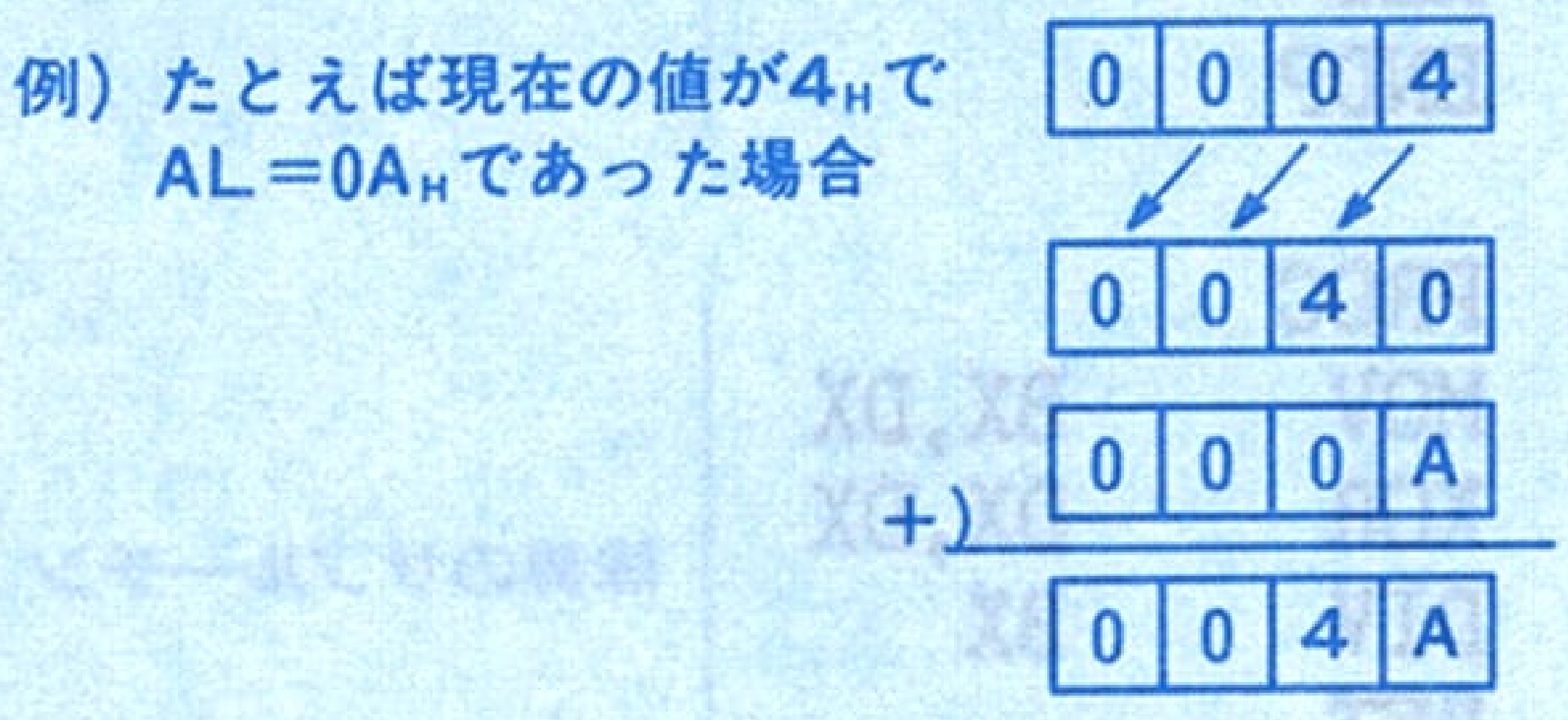
```
                    16進文字列↔バイナリ数値変換モジュール
;; HEX.ASM          ASCII String<->Binary data
;;                   Convert function

           PUBLIC   HEXTOBIN,BINTOHEX ……この2つのサブルーチンを他のモジュールで参照することを許可する宣言

CODE       SEGMENT ……………論理セグメント「CODE」の始まり
           ASSUME   CS:CODE,DS:CODE ……………セグメントレジスタが「CODE」の始まりにセットされるように宣言する

HEXTOBIN   PROC ……………16進文字列→バイナリ数値変換サブルーチンの始まり
           XOR      DX,DX ……………このルーチンの中で、中間結果はDXレジスタに保存される
           XOR      AH,AH ……………AHレジスタは、このルーチンの中では00Hである
           MOV      CX,4 ……………最大4桁
```

— リスト 5.3 —（次ページ以下に続く）



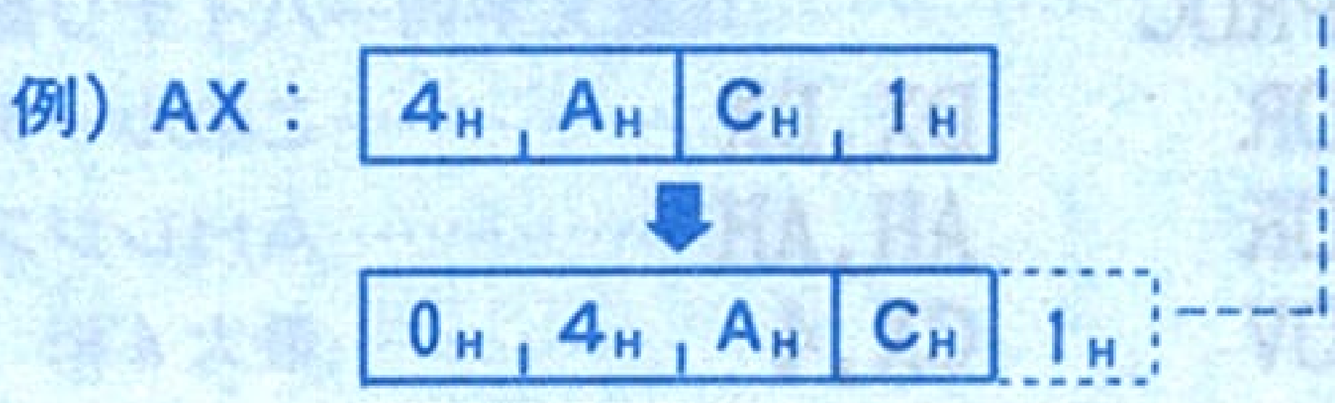
入力条件………SI：16進文字列の先頭アドレス
出力条件………AX：変換結果

```
HBNEXT:
          MOV       AL,[SI]         } 1文字取り出す
          INC       SI
          CMP       AL,'0'
          JL        HBEND
          CMP       AL,'9'          } '0'≦AL≦'9'かどうか
          JG        HBALPHA
          SUB       AL,'0'  ……'0'～'9'なら'0'を引けばその値
          JMP       SHORT HBADD ……………桁加算へ
HBALPHA:
          CMP       AL,'A'
          JL        HBEND
          CMP       AL,'F'
          JLE       HBHEX           "A≦AL≦'F'
          CMP       AL,'a'          "a≦AL≦'f'  かどうか、どちらの範囲にもない文字があればそこで終わる
          JL        HBEND
          CMP       AL,'f'
          JG        HBEND
HBHEX:
          AND       AL,5FH  ……………大文字にする
          SUB       AL,'A'-0AH ……………'A'を引いて0AHを足せば求める値になる
HBADD:
          SHL       DX,1
          SHL       DX,1
          SHL       DX,1            } 左へ4回シフトして加える
          SHL       DX,1
          ADD       DX,AX
          LOOP      HBNEXT
HBEND:
          MOV       AX,DX  ……………結果をAXレジスタに入れて返す
          RET
HEXTOBIN ENDP
```



```
BINTOHEX PROC ……………バイナリ数値→16進文字列変換サブルーチンの始まり
          ADD       SI,3 ……………下の桁から変換していく
          MOV       CX,4 ……………4桁の文字列にする
BHNEXT:
          MOV       DL,AL
          AND       DL,0FH ……………下位4ビットをとる。0FHでマスク
          CMP       DL,0AH          } 0AH以上ならその処理へ
          JGE       BHHEX
          ADD       DL,'0'          } 0AH未満なら'0'を加えるだけ
          JMP       SHORT BHSET
BHHEX:
          ADD       DL,'A'-0AH ····0AH以上なら'A'−0AHを加える
BHSET:
          MOV       [SI],DL ……………得られた文字列を格納する
          DEC       SI ……………ポインタを−1する(下の桁から上の桁へ)
          SHR       AX,1
          SHR       AX,1            } 4回右へシフトする。
          SHR       AX,1              (4ビットずつ変換すればよいため)
          SHR       AX,1
```

バイナリ数値→16進文字列
入力条件………{AX：値
SI：変換した文字列を格納するアドレス
出力条件………SI：4桁の16進文字列の格納アドレス



```
        LOOP    BHNEXT ……………4回のループで終了
        INC     SI ……………1回戻りすぎているので戻す
        RET
BINTOHEX ENDP

CODE    ENDS ……………論理セグメント「CODE」の終わり
        END ……………モジュールの終わり
```

リスト 5.3　16 進アスキー文字列↔バイナリ数値変換モジュール HEX.ASM

5.4 アセンブル

ソースファイルが書き上がると、いよいよ MASM によるアセンブルです。ここでは MASM の対話形式のコマンドにより実行してみましょう（図 5.22 参照）。なお、1 章では対話形式によらず、コマンドラインで指示する形式で実行しています。

まず、

```
A>MASM ↵
```

とのみ入力してアセンブラを起動すると、次の 4 つのファイル名を順番に問い合わせてきます。

(1) Source　filename　[.ASM]：
ソースファイルのファイル名を指定する。ファイルタイプを省略すると「.ASM」が自動的に付けられる。

(2) Object　filename　[*x*.OBJ]：
生成されるオブジェクトファイルのファイル名を指定する。単にリターンキーを入力すると、(1) で指定したファイル名がオブジェクトファイル「.OBJ」として自動的に指定される。

(3) Source　listing　[NUL.LST]：
生成されるソースリスティングファイルのファイル名を指定する。単にリターンキーを入力すると何も作成されない。

(4) Cross-reference　[NUL.CRF]：
生成されるクロスリファレンスファイルのファイル名を指定する。単にリターンキーを入力すると何も作成されない。クロスリファレンスとは、ソースファイル内で使われている名前

がどこで定義され、どこで参照されているかを表にしたもの。ここで出力されるファイルは、そのままの形では表示できないが、付属のユーティリティプログラムCREFによってリスティングファイルに変換され、TYPEコマンドなどで表示することができる。

```
A>MASM↵
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.

Source filename [.ASM]: CALC↵
Object filename [CALC.OBJ]:↵
Source listing  [NUL.LST]: CALC↵
Cross-reference [NUL.CRF]:↵

  47366 + 261638 Bytes symbol space free

     0 Warning Errors
     0 Severe  Errors

A>
```

A>MASM↵ ……パラメータなしでアセンブラを起動する

Source filename ……ソースファイルの名前を入力する。「.ASM」は省略できる

Object filename ……オブジェクトファイルの名前を入力する。ソースファイルと同じならそのままリターンキーを押す

Source listing ……リスティングファイルの名前を入力する

Cross-reference ……クロスリファレンスファイルの名前を入力する。必要なければそのままリターンキーを押す

コマンドラインでパラメータの途中まで指定することもできる。たとえばパラメータとして「A>MASM CALC↵」のように指定すると、OBJファイルのところから問い合わせが始まる。なお、対話形式のときも、以降のパラメータを省略するには「；」が使える

図5.22　アセンブラMASMの対話形式での実行例（16進電卓プログラムのメインファイルをアセンブルする）

MASMの実行は、1章で行った形式のように、コマンドラインで直接これらのパラメータを指定することもできます。各パラメータはカンマ「,」で区切って並べますが、パラメータの一部を省略することも可能です。また、途中まで指定してあとはすべて省略することも可能で、その場合には最後にセミコロン「；」を置きます。

たとえば、「A.ASM」をアセンブルして、オブジェクトファイル「A.OBJ」、リスティングファイル「A.LST」、クロスリファレンスファイル「A.CRF」を作成するには、次のコマンドラインを与えます。

```
A>MASM  A, ,A,A↵
```

また、オブジェクトファイル「A.OBJ」のみを作成するには次のコマンドラインを与えます。

```
A>MASM  A；↵
```

ですから、16進電卓プログラムでオブジェクトファイルのみを作成する場合に与えるコマンドラインは次のようになります。

```
A>MASM  CALC；↵
```

■ オプションスイッチ

MASMの実行により出力されるソースリスティングの形式を、スイッチで制御することができます。スイッチは「/ x」の形をしており、コマンドライン上のどこに指定してもかまいません。その代表的なスイッチの種類を次に示します。

/D …………パス1、パス2の両方でリスティングを作成する。パス1とパス2でオフセットが異なってしまうphaseエラーの原因を調べるのに有効。このエラーは、セグメントオーバーライド指定を付け忘れたときなどに発生する

/ML…………すべての名前に関して、大文字と小文字を区別する

/Z …………ディスプレイにエラー行を表示する

/ZI …………行番号とシンボル情報がオブジェクトファイルに出力される。CODEVIEWでソースコードデバッグを行うときには必要

/ZD …………行番号がオブジェクトファイルに出力される。SYMDEBでソースファイルを参照するときには必要

5.5 リンク

アセンブルが成功すれば次はリンクの作業です。リンクはいくつかのリロケータブル・オブジェクトファイル「.OBJ」を結合して、1本の実行可能オブジェクトファイル「.EXE」を作成するものです。さきに述べたように、ソースファイルが1つしかない場合でもリンクは必要です。

リンクは、MS-DOSの標準リンカLINKによって実行します。ここでのLINK作業は、アセンブルと同じように対話形式で行ってみましょう。なお、1章では対話形式によらず、コマンドラインで指示する形式で実行しています。

```
A>LINK⏎
```

とのみ入力してリンカを起動すると、次の4つのファイル名を順番に問い合わせてきます。

(1) Object Modules [.OBJ]:

アセンブラから出力されたオブジェクトファイル「.OBJ」のファイル名を、「+」あるいは空白で区切って、結合する順に並べる。1行に書ききれない場合は、行の最後のファイル名を「+⏎」で終わると、その続きを再度問い合わせてくる。

(2) Run　File　[*x*.EXE]：

生成される実行可能ファイル「.EXE」のファイル名を指定する。単にリターンキーを入力すると、(1)の先頭に指定したオブジェクトファイル名が自動的に指定される。

(3) List　File　[NUL.MAP]：

論理セグメントの配置に関する情報を知るために生成されるリスティングファイルのファイル名を指定する。必要のない場合は、単にリターンキーを入力すれば自動的にヌルファイル「NUL.MAP」が指定され、この出力は捨てられる。

(4) Libraries　[.LIB]：

結合するライブラリのモジュール名「.LIB」のリストを、(1)と同様に「＋」または空白で区切って並べる。ライブラリに関して詳しくは本章の5.7で解説する。

以上は対話形式で実行する場合ですが、このLINKもMASMと同様に、コマンドラインで直接これらのパラメータを指定することもできます。パラメータは「,」で区切って省略することもでき、また「；」以降のパラメータがすべて省略されることなどもMASMの場合と同じです。

たとえば、オブジェクトファイル、「A.OBJ」「B.OBJ」「C.OBJ」と、ライブラリファイル「MYLIB.LIB」をリンクして、実行可能オブジェクトファイル「Z.EXE」を作成する場合は、次のコマンドラインを実行します。

```
A>LINK  A  B  C  Z, ,MYLIB⏎
```

また、実行可能オブジェクトファイルを最初のモジュールの名前「A」と同じにする場合は、

```
A>LINK  A  B  C, , ,MYLIB⏎
```

となります。たとえば、例題の16進電卓プログラムの場合は次のとおりです。

```
A>LINK  CALC  OPE  HEX ;⏎
```

■ オブジェクト形式の変換

本章の5.2で述べたように、EXE形式からCOM形式へ、オブジェクトファイルの形式を変換するには、MS-DOS標準のオブジェクト形式変換プログラムEXE2BINを実行します。さきに述べたように、EXEファイルには、リロケーション情報が含まれています。COMファイルには、ロード時のリロケートの必要はありませんので、EXE2BINにより、それらの不要な情報を削除します。

EXE2BINの実行によって変換しようとするEXEファイルは、次の条件を満足していなければなりません。

- プログラム開始アドレスがEND擬似命令によって指定されており、そのプログラム先頭のセグメントのオフセットは0100_Hであること
- ロード時のリロケートが必要となるセグメントの参照命令を含まないこと
- スタックセグメントがないこと

EXE2BINを実行するためのコマンドラインを次に示します。このコマンドラインにより、A.EXEからA.COMが生成されます。

```
A>EXE2BIN　A　A.COM⏎
```

コマンドラインの最初のパラメータは、EXEファイルのファイル名を指定し、そのファイルタイプ「.EXE」は省略できます。2番目のパラメータは、変換後のファイル名を指定します。ただし、このファイルタイプ「.COM」を省略すると、「.BIN」が自動的に付けられますので、実際に実行するプログラムの場合は、必ず「.COM」を指定しなければなりません。

また、EXE2BINは、セグメントの参照を含むEXEファイルについても、セグメントの絶対アドレスを指定することにより、バイナリファイルに変換する機能を持っています。この機能は、プログラムをROM化する場合などに利用することができます。

5.6　デバッグ

リンクの作業が終わり、(COM形式のプログラムであればEXE2BINを実行して)実行可能なオブジェクトプログラムができあがると、そのプログラムを実際に実行して動作テストを行います。さて、期待どおりの動作をするでしょうか。現実には、いくら優秀なプログラマーの手によるものでも、1回で完動することはまずあり得ません。とくにアセンブラでの開発は、たいていは暴走してリセットボタンを押すはめになるでしょう。

だからといって、がっかりする必要はありません。動かないのはあたり前のことで、できあがったプログラムには、必ずバグがあるものです。また、正常に動作しているように見えても、ほかの動作をさせるとダメになることもあります。プログラムの開発過程でデバッグ作業の占める割合は大きく、大半はデバッグ作業に費やされるといってもよいほどです。

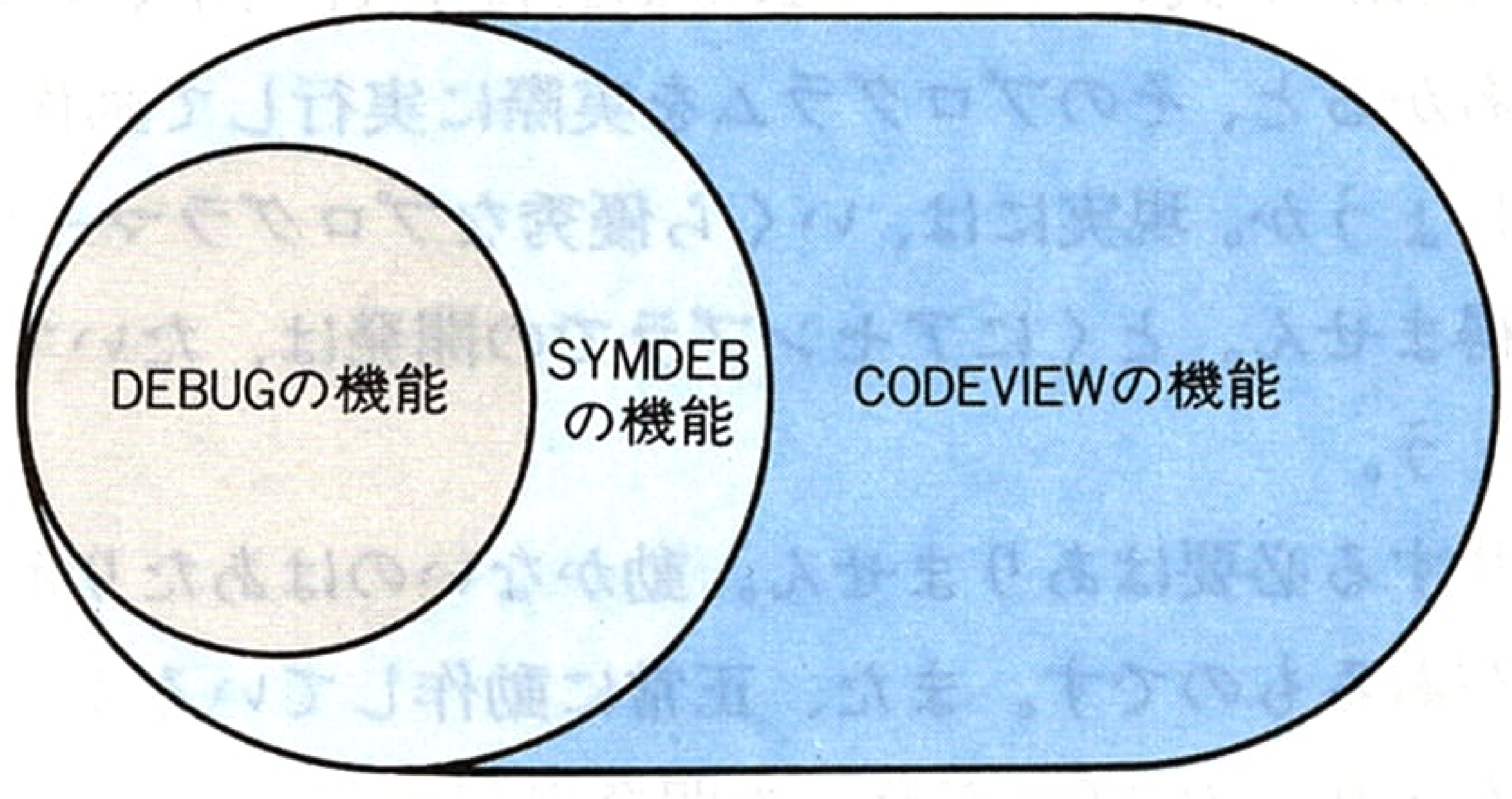
さて、実際にデバッグを行う際のバグ攻略法については、やはり経験的に学ぶ必要があります。ただし、その経験の上に臨機応変で柔軟な思考が非常に大切です。プログラマーの知恵が輝くのは、このデバッグのときかも知れません。次に、一般的なデバッグ時の考え方の手順を示しておきましょう。

① 症状をじっくり観察する ………………プログラムをいろいろな角度から実行した場合の症状をよく見ておくことが大切。推理するための重要な手掛かりになる
② ソースファイルを確かめる ……………バグの多くは単純なタイプミスである
③ アルゴリズムを検証する ………………プログラムを作る前に行うものだが、アルゴリズムに矛盾があるかもしれない

どのようなプログラミング言語を使っても、デバッグ作業は、基本的にはこの繰り返しです。③のアルゴリズムの検証については、プログラムの流れを「人間コンピュータ」になって、机上で追跡すればよいのですが、これはなかなかたいへんな作業です。高級言語であれば、たとえば、変数の値を表示する文を要所要所に挿入するなどのテクニックが簡単に使えますが(6章図6.5参照)、アセンブラではそれもたいへんです。そこで**デバッガ**の助けが必要になります。

MS-DOS標準のデバッガは**SYMDEB**(シンボリックデバッガ：後述)ですが、これはMS-DOSのバージョン3.xに付属しているもので、それ以前は**DEBUG**というプログラム名でした。SYMDEBはDEBUGのアッパーコンパチブルのデバッガですので、DEBUGのコマンドはそのままSYMDEBで実行できます(**図5.23**参照)。

また、ソースコードデバッガと呼ばれるCODEVIEW(MASMバージョン5.1以降やMS-Cバージョン4.0以降に付属している)は、さらに大幅に機能を強化させたデバッガになっています(後述)。なお、CODEVIEWは、とくにC言語などの高級言語のデバッグに威力を発揮します。



＊ただし、CODEVIEWはSYMDEB、DEBUGの機能のすべてを
包括しているわけではない

▢部はSYMDEBになって追加された機能
🟦部はCODEVIEWで追加された機能

図5.23　DEBUG、SYMDEB、CODEVIEWの関係

これらのデバッガは、多くの強力な機能を持ったなかなか優秀なもので、ソフトウェア開発者はこれを自在に使いこなすことを要求されます。

デバッグ作業は、プログラムの途中にブレークポイントというものを設定して、そこでプログラムの実行を一時中断させ、その時点でのCPUの各レジスタの値を表示させたり、1命令ずつレジスタの値を確認しながら実行させたり、レジスタやメモリの値を書き換えて実行するなど、そのほかにも実にいろいろなことが可能です。またデバッガは、2章で行ったように、ディスクのセクタのデータを直接読み書きするなど、デバッグ以外の目的にも使います。

表5.2にSYMDEB内のコマンド機能の一覧表を示しましょう。

コマンド名	機　能	コマンド名	機　能
A	アセンブル	N	FCBに名前をセットする
BP	ブレークポイントの設定	XO	カレント・シンボルマップ、セグメントをセットする
BC	ブレークポイントの解除		
BD	ブレークポイントの一時的無効化	O	I/Oポートに出力
BE	ブレークポイントの有効化	P	割り込みにも対応したトレース
BL	ブレークポイントの情報の表示	Q	終了
*	コメント	<、>、= {、}、~	標準入出力の切り換え
C	2つのメモリ領域の比較		
?	値の表示	R	レジスタの内容の表示、設定
D	メモリのダンプ	S	バイト値の検索
E	メモリ内容の変更	S	ソースモードの設定
X	シンボルの情報の表示	!	MS-DOSコマンドの実行
F	メモリを指定された値で満たす	.	カレント・ソースラインの表示
G	プログラムの実行。ブレークポイントの設定	K	スタックフレームの情報の表示
?	ヘルプ	Z	シンボルに値をセットする
H	2つの値の和と差の計算	T	トレース
I	I/Oポートからの入力	U	逆アセンブル
L	ディスクからの読み出し	V	ソースラインの表示
M	メモリ内容の転送	W	ディスクへの書き込み

（『MS-DOS 3.1ハンドブック』（アスキー出版局刊）より）

表5.2　SYMDEBコマンド機能一覧

次に、デバッグ時によく使われる代表的なSYMDEB（DEBUG）内のコマンドと、その基本的な使い方を示します。これらのデバッガは、

```
A>SYMDEB↵　あるいは　A>DEBUG↵　（以降DEBUGについては省略）
```

によって起動します。ターゲットプログラム（デバッグの対象となるプログラム）を指定していないので、ターゲットプログラムはロードされていませんが、SYMDEB内のコマンドは使うことができます（もちろんこの状態から任意のターゲットプログラムをロードすることも可能）。

```
A>SYMDEB　〈ファイル名〉　[パラメータ]↵
```

このように、ターゲットプログラムのファイル名を指定してSYMDEBを起動すると、SYMDEBが起動された上で、あたかもCOMMAND.COM上からターゲットプログラムを、

```
A>〈ファイル名〉　[パラメータ]↵
```

のように実行したと同じような環境が実現されます。つまり、SYMDEB上でターゲットプログラムを実際に実行しながらデバッグができるのです。この[パラメータ]は、ターゲットプログラムの実行に関するパラメータです。たとえば、1章のCHMODプログラムをデバッグする場合は、

```
A>SYMDEB　CHMOD.COM　FORMAT.EXE/R↵
```

のようにします。この場合は、CHMODプログラムがSYMDEBによって実行され、ディスクフォーマットプログラムのFORMAT.EXEが、リードオンリーファイルに設定される過程をデバッグすることができます。SYMDEBは、COMMAND.COMがCHMODプログラムを実行する場合と同じように、ターゲットプログラムが実行されるための環境を整えてロードします。ターゲットプログラム側から見れば、COMMAND.COMによって通常実行される場合とまったく同じ状況です。

さて、上記のコマンドラインを実行してSYMDEBの世界にはいると、SYMDEBのプロンプト「－」が表示され、デバッグ内の各種のコマンドが使えます*。以下に、それらの代表的なものを示しましょう。なお、その際、SYMDEB内では、表示される数値や入力する数値は、2進数、8進数、10進数、16進数で入力することができます（DEBUGは、16進数のみ）。また、ターゲットプログラムを指定してSYMDEBを起動したときには、IPレジスタはCOM形式の場合は0100Hにセットされ、EXE形式の場合は実行開始アドレスにセットされています。ですから、以下に述べるD、G、T、Uなどのコマンドの場合、IPレジスタの指定を省略すると、最初に実行する際には、通常、プログラムの先頭アドレスから実行が開始されることになります。

---

* SYMDEBでは各種のコマンドあとに、スペースを置かなければならない（DEBUGでは、どちらでもよい）。たとえばDコマンドでは、「-D　100　1FF↵」のように入力する。

Dump —— 指定した範囲のアドレスのメモリ内容をダンプする

書式　D　[〈開始アドレス〉　〈終了アドレス〉]↵
D　[〈開始アドレス〉　L〈バイト数*〉]↵

Dコマンドは、具体的には次のようなコマンドラインで実行します(図5.24参照)。

－D　100　1FF↵ ………アドレス0100$_\mathrm{H}$～01FF$_\mathrm{H}$の範囲がダンプされる
－D　100　L100↵………アドレス0100$_\mathrm{H}$から100$_\mathrm{H}$バイト(256バイト)分がダンプされる。つまり0100$_\mathrm{H}$～01FF$_\mathrm{H}$の範囲がダンプされる(上と同じ結果)
－D　100↵ ………………0100$_\mathrm{H}$から80$_\mathrm{H}$バイト(128バイト)分がダンプされる
－D↵………………………直前に実行したDコマンドの続きから128バイト分がダンプされる

Enter —— メモリの内容を1バイトずつ書き換えたり表示したりする

書式　E　〈アドレス〉　[〈値1〉　〈値2〉　〈値3〉…]↵

Eコマンドは、具体的には次のようなコマンドラインで実行します(図5.24参照)。

－E　500　03　BE　4D↵ ………アドレス0500$_\mathrm{H}$からのメモリ内容を、03$_\mathrm{H}$、BE$_\mathrm{H}$、4D$_\mathrm{H}$に書き換える
－E　500↵ …………………………〈値〉を省略すると、指定したアドレスのメモリ内容が表示され、そのあとに書き換える値を入力していく

Go —— ターゲットプログラムを実行する

書式　G[＝〈実行開始アドレス〉][〈ブレークポイントアドレス1〉　〈ブレークポイントアドレス2〉…]↵

Gコマンドは、具体的には次のようなコマンドラインで実行します(図5.24参照)。

－G＝500　600　650↵ …………ブレークポイントを0600$_\mathrm{H}$と0650$_\mathrm{H}$に置いて、プログラムを0500$_\mathrm{H}$から実行する
－G＝500↵…………………………プログラムを0500$_\mathrm{H}$から実行する
－G↵………………………………現在のCS：IPレジスタが示すアドレスから実行する

＊この「バイト数」は、SYMDEBでは場合によってバイト数、ワード数、ダブルワード数、浮動小数点数など、直前に実行したDコマンドの指定と同じになる。

「＝〈実行開始アドレス〉」によって、実行開始アドレスを指定します。なお、現在のCS：IPレジスタの値から実行する場合には省略できます。〈ブレークポイントアドレス〉を指定すると、そのアドレスにブレークポイントがセットされます。ブレークポイントをセット(最大10か所)しておくと、そのアドレスの命令を実行しようとする直前で実行が停止します。つまり、プログラムを任意のアドレスまで実行することができるのです。ブレークポイントで実行が停止すると、そのときの各レジスタの値が表示されます。もちろん、必要があればGコマンドによってそのあとのプログラムの実行を続けることができます。

Register —— 各レジスタの内容を表示あるいはセットする

書式　R　[〈レジスタ名〉]⏎

Rコマンドは、具体的には次のようなコマンドラインで実行します(図5.24参照)。

−R　AX⏎ ………現在のAXレジスタ(レジスタは任意)の値を表示し、AXレジスタにセットする値の入力待ちとなる。変更しない場合はキャリジリターンを入力する

−R　F⏎ …………現在のすべてのフラグの状態を表示し、各フラグにセットする値(状態)の入力待ちとなる。変更しない場合はキャリジリターンを入力する

−R⏎………………現在のすべてのレジスタの値とフラグの状態を表示する

フラグ名	セット「1」	リセット「0」	フラグ名	セット「1」	リセット「0」
オーバーフロー	OV(オーバー)	NV(ノンオーバー)	ゼロ	ZR(ゼロ)	NZ(ノンゼロ)
ディレクション	DN(減少)	UP(増加)	補助キャリー	AC(キャリー)	NA(ノンキャリー)
割り込み	EI(許可)	DI(禁止)	パリティ	PE(偶数)	PO(奇数)
サイン	NG(負)	PL(正)	キャリー	CY(キャリー)	NC(ノンキャリー)

表5.3　SYMDEBのRコマンドでフラグの設定に使われる状態コード

Trace —— プログラムを1ステップ(1命令)ずつ実行しながら、そのつど各レジスタの値やそのステップの命令のニーモニックを表示する

書式　T [＝〈開始アドレス〉]　[〈ステップ数〉] ⏎

Tコマンドは、具体的には次のようなコマンドラインで実行します(図5.24参照)。

−T=500　1F⏎…………アドレス0500Hから1FH(31)ステップ分、トレースを実行する

−T=500⏎………………0500Hから1ステップのみトレースを実行する

－T　F↵ ……………………現在のCS：IPレジスタからF$_{\mathrm{H}}$(15)ステップ分トレースを実行する。最初の実行は、プログラムの先頭から行われる
－T↵…………………………現在のCS：IPレジスタから1ステップのみトレースを実行する

この各レジスタの変化を1命令ずつ見ながらプログラムを実行するTコマンドの機能を、**トレース**と呼びます。このトレースは、デバッグにおいて非常に強力な機能であり、CPUの動きを実際に目で追いながらプログラムをテストすることができます。なお、このTコマンドを実行する際には、Gコマンドと組み合わせて、必要な部分だけトレースし、あとはGコマンドで実行すると効率よくデバッグを行うことができるでしょう。とくに詳しく調べたいところの直前までGコマンドで実行し、そこから1ステップずつトレースすれば効果的です。

Tコマンドは、8086系CPUのトレース割り込み機能を利用しているので、ROM内のプログラムであってもトレースすることが可能です。ただし、このTコマンドには注意しなければならないことが1つあります。8086系CPUはパイプライン処理を行っているので、1ステップのみ実行しようと思っても、数命令実行されることがありますが、トレースの表示上では、ある命令が表示されずに飛ばされて、実行されなかったかのように見えるのです。このような場合でも、レジスタの値などを見れば実行されていることがわかりますが、うっかり見逃してしまうこともありますから、トレースする際には、ソースリストと突き合わせながら、ときどきUコマンドで逆アセンブルするなどして確認してください。

Unassemble —— 指定した範囲のメモリ内容を逆アセンブルする

書式　U　[〈開始アドレス〉　〈終了アドレス〉]↵
　　　　U　[〈開始アドレス〉　L〈バイト数〉]↵

範囲の指定やコマンドの使い方はDコマンドと同じです。ただし、範囲の指定がない場合は、20_{H}(32)バイト分が逆アセンブルされます(図5.24参照)。

さきのDコマンド、およびこのUコマンドの2つは、メモリ内容が期待どおりのコードやデータであるかどうかを確認するために使います。これらのコマンドによって、文法や表記法の誤りから、思わぬ結果になっていることを発見することもあります。

Quit —— SYMDEBを終了する

書式　Q↵

SYMDEBを終了し、MS-DOSに戻ります。

さて、デバッグ作業は、以上のコマンドの中から、「D：ダンプ」、「G：実行」、「T：トレース」、「U：逆アセンブル」を使いこなすだけでも、十分プログラムの流れを追うことができます。さらに「E：エンター」、「R：レジスタ」を使えば、かなり本格的なデバッグを行うことも可能です。デバッガを使う上で、何より大切なことは、これらのコマンドを効果的に使い分けることです。ただし、これにはかなり高度な知識と経験が必要となるでしょう。

以上解説したSYMDEBの主なコマンドの実行例を図**5.24**に示します。

```
A>SYMDEB CALC.COM↵ ……………SYMDEBを起動すると同時に、ターゲットプログラム「CALC.COM」をロードする
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-D 100↵ ……………アドレス0100Hから128バイト分ダンプする
3AE6:0100  33 C0 A3 EE 01 B0 3D A2-F0 01 B4 02 B2 23 CD 21  3@#n.0="p.4.2#M!
3AE6:0110  8D 36 F1 01 B9 04 00 E8-A1 00 8A D8 83 F9 04 74  .6q.9..h!..X.y.t
3AE6:0120  F6 E8 84 00 75 F1 EB 16-8D 36 F1 01 B9 04 00 E8  vh..uqk..6q.9..h
3AE6:0130  89 00 8A D8 E8 71 00 75-F6 83 F9 04 74 F1 8A D3  ...Xhq.uv.y.tq.S
3AE6:0140  E8 A6 00 C6 04 24 8D 36-F1 01 E8 C3 00 8B 16 EE  h&.F.$.6q.hC...n
3AE6:0150  01 92 86 1E F0 01 80 FB-2B 74 13 80 FB 2D 74 13  ....p..{+t..{-t.
3AE6:0160  80 FB 2A 74 13 80 FB 2F-74 13 8B C2 EB 12 E8 8F  .{*t..{/t..Bk.h.
3AE6:0170  00 EB 0D E8 8D 00 EB 08-E8 8B 00 EB 03 E8 89 00  .k.h..k.h..k.h..
-D↵ ……………その続きを128バイト分ダンプする
3AE6:0180  A3 EE 01 8A 16 F0 01 80-FA 3D 75 9C 8D 36 F1 01  #n...p..z=u..6q.
3AE6:0190  E8 B6 00 B4 09 8D 16 F1-01 CD 21 B2 0D E8 49 00  h6.4...q.M!2.hI.
3AE6:01A0  B2 0A E8 44 00 E9 58 FF-3C 3D 74 0E 3C 2B 74 0A  2.hD.iX.<=t.<+t.
3AE6:01B0  3C 2D 74 06 3C 2A 74 02-3C 2F C3 E8 26 00 3C 30  <-t.<*t.</Ch&.<0
3AE6:01C0  7C 21 3C 39 7E 10 3C 41-7C 19 3C 46 7E 08 3C 61  |!<9~.<A|.<F~.<a
3AE6:01D0  7C 11 3C 66 7F 0D E3 E3-88 04 46 49 8A D0 E8 08  |.<f..cc..FI.Ph.
3AE6:01E0  00 EB D8 C3 B4 08 CD 21-C3 B4 02 CD 21 C3 00 00  .kXC4.M!C4.M!C..
3AE6:01F0  3D 00 00 00 00 24 00 00-00 00 00 00 00 00 00 00  =....$..........
-E 100↵ ……………アドレス0100Hからのデータを順に書き換える
3AE6:0100  33.01␣  C0.02␣  A3.03␣  EE.↵ ←――最後はリターンで終わる
```

その時点の内容が1バイトだけ表示される

そのバイトのデータを書き換える場合は任意のデータを入力してスペースを入力する

そうすると次のバイトのデータが表示される。その繰り返し。書き換えない場合はスペースのみ入力する

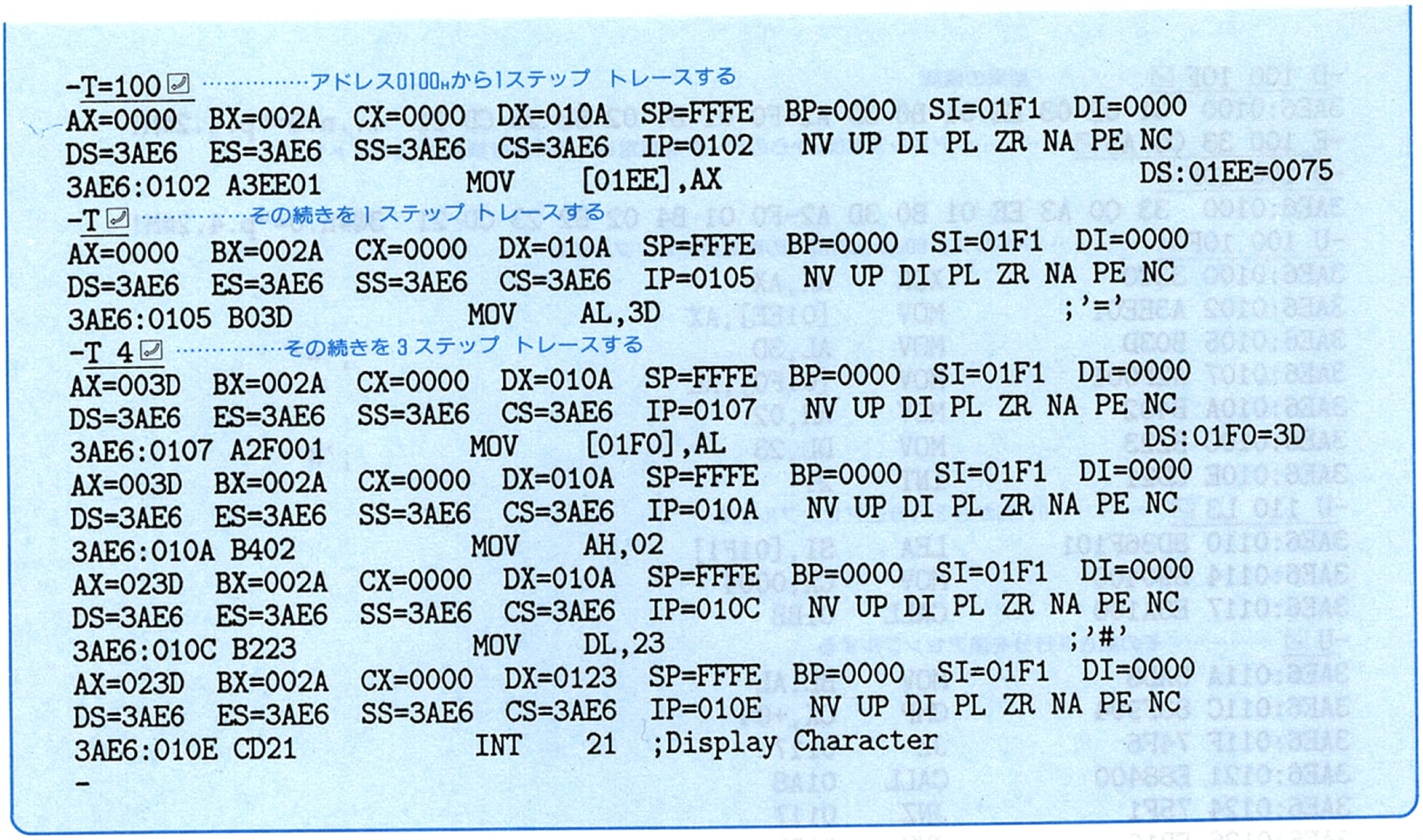
― 図 5.24 ―（次ページ以下に続く）

```
-D 100 10F⏎ ……………結果の確認
3AE6:0100  01 02 03 EE 01 B0 3D A2-F0 01 B4 02 B2 23 CD 21   ...n.0="p.4.2#M!
-E 100 33 C0 A3⏎ ……………アドレス0100Hからのデータを指定した値に書き換える(3バイト分)
-D 100 10F⏎
3AE6:0100  33 C0 A3 EE 01 B0 3D A2-F0 01 B4 02 B2 23 CD 21   3@#n.0="p.4.2#M!
-U 100 10F⏎ ……………アドレス0100Hから010FHの間を逆アセンブルする
3AE6:0100 33C0          XOR     AX,AX
3AE6:0102 A3EE01        MOV     [01EE],AX
3AE6:0105 B03D          MOV     AL,3D                           ;'='
3AE6:0107 A2F001        MOV     [01F0],AL
3AE6:010A B402          MOV     AH,02
3AE6:010C B223          MOV     DL,23                           ;'#'
3AE6:010E CD21          INT     21
-U 110 L3⏎ ……………0110Hから3行分逆アセンブルする
3AE6:0110 8D36F101      LEA     SI,[01F1]
3AE6:0114 B90400        MOV     CX,0004
3AE6:0117 E8A100        CALL    01BB
-U⏎ ……………その続き8行分を逆アセンブルする
3AE6:011A 8AD8          MOV     BL,AL
3AE6:011C 83F904        CMP     CX,+04
3AE6:011F 74F6          JZ      0117
3AE6:0121 E88400        CALL    01A8
3AE6:0124 75F1          JNZ     0117
3AE6:0126 EB16          JMP     013E
3AE6:0128 8D36F101      LEA     SI,[01F1]
3AE6:012C B90400        MOV     CX,0004
-G=100 126⏎ ……………ブレークポイントを0126Hに設定して0100Hから実行する
#4FAX=082F  BX=002F  CX=0002  DX=0046  SP=FFFE  BP=0000  SI=01F3  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=0126   NV UP EI PL ZR NA PE NC
3AE6:0126 EB16          JMP     013E
  CALCプログラムが起動したので「4」「F」「/」と入力した。「4」=34H、「F」=46H、「/」=2FH。「/」は表示されていない
-BP 100⏎ ……………ブレークポイントを0100Hに設定する
-G⏎ ……………続くプログラムを実行する
/2*3=0075
AX=020A  BX=002A  CX=0000  DX=010A  SP=FFFE  BP=0000  SI=01F1  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=0100   NV UP EI PL NZ NA PO NC
3AE6:0100 33C0          XOR     AX,AX                         ;BR0
-R AX⏎ ……………現在のAXレジスタの値を表示する
AX 020A ……………020AHであった
:0000⏎ ……………「:」が表示されているので、そのあとにセットする値を入力する(0000Hを入力した)
-R F⏎ ……………現在のすべてのフラグの状態を表示する
NV UP EI PL NZ NA PO NC -DI
-R⏎ ……………現在のすべてのレジスタの状態を表示する
AX=0000  BX=002A  CX=0000  DX=010A  SP=FFFE  BP=0000  SI=01F1  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=0100   NV UP DI PL NZ NA PO NC
                                              「0」「0」「0」「0」「0」
3AE6:0100 33C0          XOR     AX,AX                         ;BR0
                                                              「0」「0」「0」
```

現在のIP=0100Hにあるオブジェクトコードとそのニーモニックが表示される

リセットされた

ここにブレークポイントの0番が設定されていることを示している



```
-T=100⏎ ……………アドレス0100Hから1ステップ トレースする
AX=0000  BX=002A  CX=0000  DX=010A  SP=FFFE  BP=0000  SI=01F1  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=0102   NV UP DI PL ZR NA PE NC
3AE6:0102 A3EE01          MOV     [01EE],AX                    DS:01EE=0075
-T⏎ ……………その続きを1ステップ トレースする
AX=0000  BX=002A  CX=0000  DX=010A  SP=FFFE  BP=0000  SI=01F1  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=0105   NV UP DI PL ZR NA PE NC
3AE6:0105 B03D            MOV     AL,3D                     ;'='
-T 4⏎ ……………その続きを3ステップ トレースする
AX=003D  BX=002A  CX=0000  DX=010A  SP=FFFE  BP=0000  SI=01F1  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=0107   NV UP DI PL ZR NA PE NC
3AE6:0107 A2F001          MOV     [01F0],AL                    DS:01F0=3D
AX=003D  BX=002A  CX=0000  DX=010A  SP=FFFE  BP=0000  SI=01F1  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=010A   NV UP DI PL ZR NA PE NC
3AE6:010A B402            MOV     AH,02
AX=023D  BX=002A  CX=0000  DX=010A  SP=FFFE  BP=0000  SI=01F1  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=010C   NV UP DI PL ZR NA PE NC
3AE6:010C B223            MOV     DL,23                     ;'#'
AX=023D  BX=002A  CX=0000  DX=0123  SP=FFFE  BP=0000  SI=01F1  DI=0000
DS=3AE6  ES=3AE6  SS=3AE6  CS=3AE6  IP=010E   NV UP DI PL ZR NA PE NC
3AE6:010E CD21            INT     21  ;Display Character
-
```

図 5.24　SYMDEB のコマンド実行例

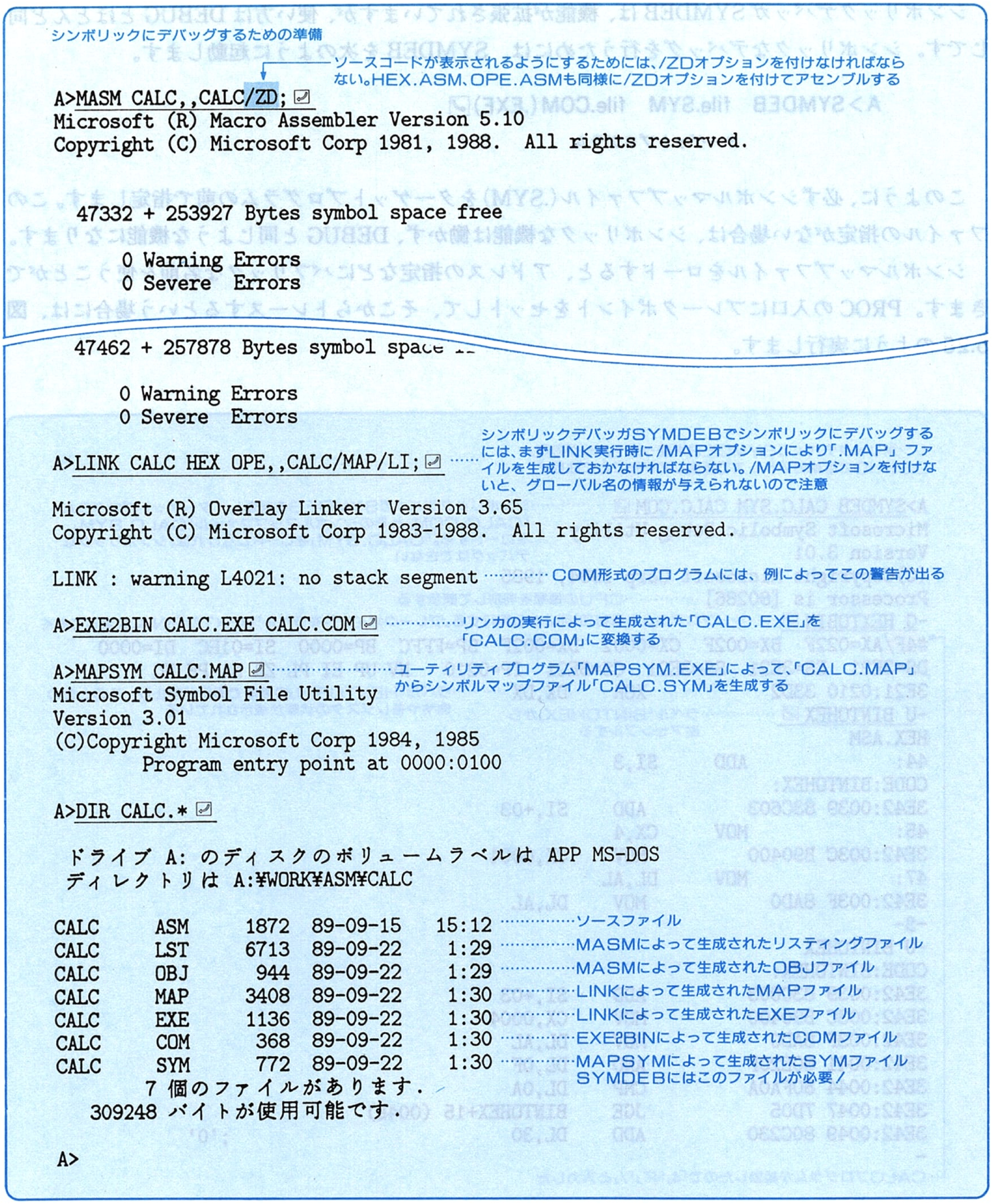
■ シンボリックデバッガ

SYMDEB をシンボリックデバッガとして使うことにより、シンボルを使ったデバッグができます。「シンボリック」とは、デバッガ内の各種のコマンドにおいて、そのアドレスの指定などに、ソースファイルで使っているラベル名などを使用できることを指します。この機能は実に便利であり、ブレークポイントの指定やメモリのダンプなどで、以前は絶対アドレス値で指定していたものが、ソースファイルのラベルやシンボルで指定できるために、劇的に効率が上がります。

ただし、使えるのは PUBLIC 宣言されているものだけです。それ以外は、ほかのモジュールと重複する可能性があるため当然使えません。

シンボルによるデバッグを行うには、シンボルマップファイル「.SYM」が必要です。そのためにはリンク時に/MAP オプションを指定して、パブリックシンボル・マップファイル「.MAP」を作っておきます。この/MAP オプションを付けずに作られた MAP ファイルでは機能しませんので注意してください。/MAP オプションにより作成された MAP ファイルを、MAP → SYM ファイル変換プログラム MAPSYM を実行して SYM ファイルに変換したものを使用します(図 5.25 参照)。

こうして作成された SYM ファイルが、パブリックシンボル・マップファイルと呼ばれるファイルで、PUBLIC 宣言されている名前をすべて含んでいます。



シンボリックにデバッグするための準備

```
A>MASM CALC,,CALC/ZD;↵
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


  47332 + 253927 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

```
  47462 + 257878 Bytes symbol space

      0 Warning Errors
      0 Severe  Errors

A>LINK CALC HEX OPE,,CALC/MAP/LI;↵

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

LINK : warning L4021: no stack segment

A>EXE2BIN CALC.EXE CALC.COM↵

A>MAPSYM CALC.MAP↵
Microsoft Symbol File Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
       Program entry point at 0000:0100

A>DIR CALC.*↵

 ドライブ A: のディスクのボリュームラベルは APP MS-DOS
 ディレクトリは A:¥WORK¥ASM¥CALC

CALC     ASM     1872  89-09-15   15:12
CALC     LST     6713  89-09-22    1:29
CALC     OBJ      944  89-09-22    1:29
CALC     MAP     3408  89-09-22    1:30
CALC     EXE     1136  89-09-22    1:30
CALC     COM      368  89-09-22    1:30
CALC     SYM      772  89-09-22    1:30
        7 個のファイルがあります.
    309248 バイトが使用可能です.

A>
```

ソースコードが表示されるようにするためには、/ZDオプションを付けなければならない。HEX.ASM、OPE.ASMも同様に/ZDオプションを付けてアセンブルする

シンボリックデバッガSYMDEBでシンボリックにデバッグするには、まずLINK実行時に/MAPオプションにより「.MAP」ファイルを生成しておかなければならない。/MAPオプションを付けないと、グローバル名の情報が与えられないので注意

COM形式のプログラムには、例によってこの警告が出る

リンカの実行によって生成された「CALC.EXE」を「CALC.COM」に変換する

ユーティリティプログラム「MAPSYM.EXE」によって、「CALC.MAP」からシンボルマップファイル「CALC.SYM」を生成する

ソースファイル

MASMによって生成されたリスティングファイル

MASMによって生成されたOBJファイル

LINKによって生成されたMAPファイル

LINKによって生成されたEXEファイル

EXE2BINによって生成されたCOMファイル

MAPSYMによって生成されたSYMファイル　SYMDEBにはこのファイルが必要！

図 5.25　シンボリックデバッガ SYMDEB に入力するシンボルマップファイルの作成

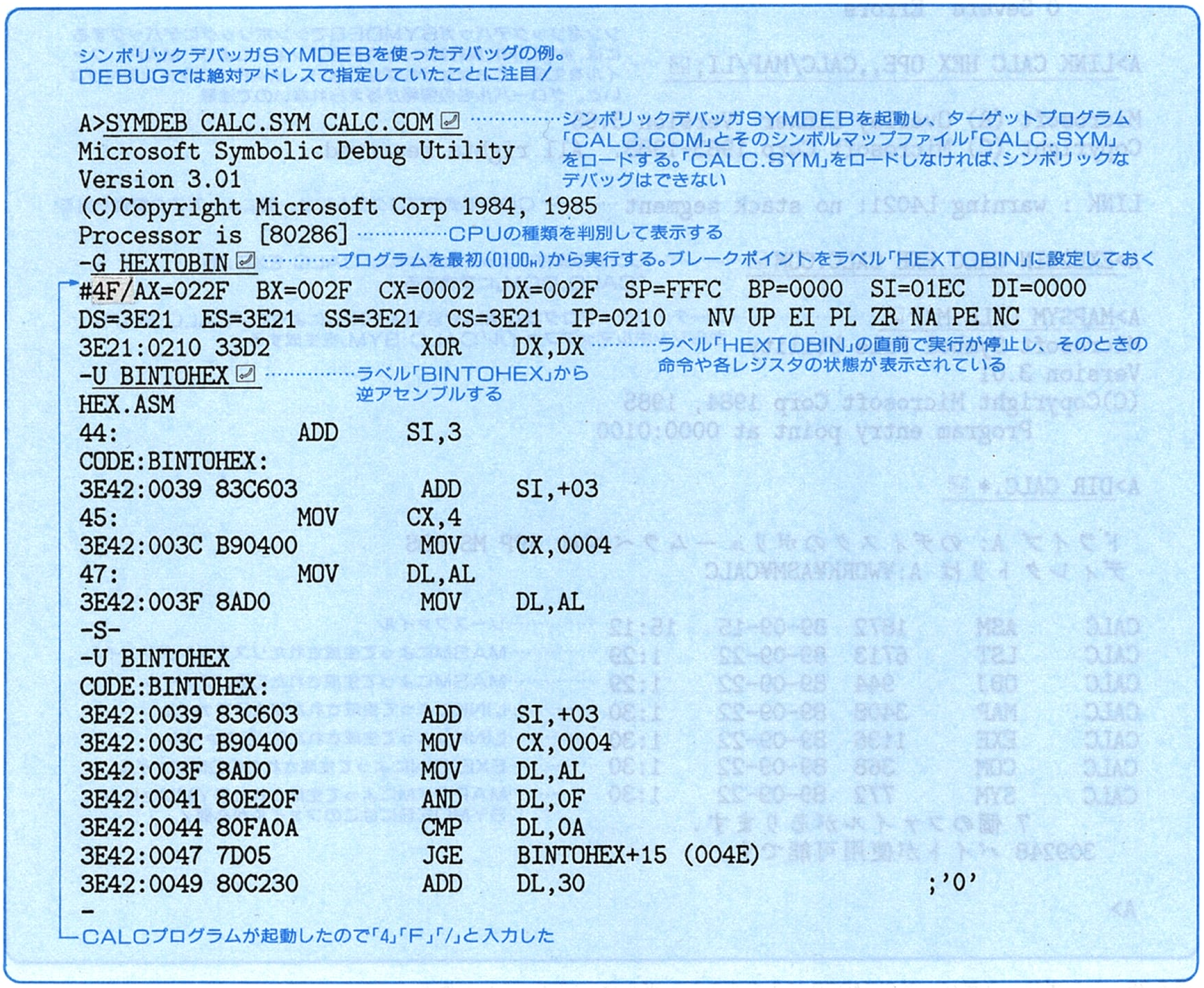
シンボリックデバッガSYMDEBは、機能が拡張されていますが、使い方はDEBUGとほとんど同じです。シンボリックなデバッグを行うためには、SYMDEBを次のように起動します。

```
A>SYMDEB  file.SYM  file.COM(.EXE)⏎
```

ターゲットプログラム

このように、必ずシンボルマップファイル(.SYM)をターゲットプログラムの前で指定します。このファイルの指定がない場合は、シンボリックな機能は働かず、DEBUGと同じような機能になります。

シンボルマップファイルをロードすると、アドレスの指定などにパブリックな名前を使うことができます。PROCの入口にブレークポイントをセットして、そこからトレースするという場合には、図5.26のように実行します。



シンボリックデバッガSYMDEBを使ったデバッグの例。
DEBUGでは絶対アドレスで指定していたことに注目／

```
A>SYMDEB CALC.SYM CALC.COM⏎
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-G HEXTOBIN⏎
#4F/AX=022F  BX=002F  CX=0002  DX=002F  SP=FFFC  BP=0000  SI=01EC  DI=0000
DS=3E21  ES=3E21  SS=3E21  CS=3E21  IP=0210   NV UP EI PL ZR NA PE NC
3E21:0210 33D2           XOR     DX,DX
-U BINTOHEX⏎
HEX.ASM
44:            ADD     SI,3
CODE:BINTOHEX:
3E42:0039 83C603         ADD     SI,+03
45:            MOV     CX,4
3E42:003C B90400         MOV     CX,0004
47:            MOV     DL,AL
3E42:003F 8AD0           MOV     DL,AL
-S-
-U BINTOHEX
CODE:BINTOHEX:
3E42:0039 83C603         ADD     SI,+03
3E42:003C B90400         MOV     CX,0004
3E42:003F 8AD0           MOV     DL,AL
3E42:0041 80E20F         AND     DL,0F
3E42:0044 80FA0A         CMP     DL,0A
3E42:0047 7D05           JGE     BINTOHEX+15 (004E)
3E42:0049 80C230         ADD     DL,30                          ;'0'
-
```

シンボリックデバッガSYMDEBを起動し、ターゲットプログラム「CALC.COM」とそのシンボルマップファイル「CALC.SYM」をロードする。「CALC.SYM」をロードしなければ、シンボリックなデバッグはできない

CPUの種類を判別して表示する

プログラムを最初(0100_H)から実行する。ブレークポイントをラベル「HEXTOBIN」に設定しておく

ラベル「HEXTOBIN」の直前で実行が停止し、そのときの命令や各レジスタの状態が表示されている

ラベル「BINTOHEX」から逆アセンブルする

CALCプログラムが起動したので「4」「F」「/」と入力した

図5.26　シンボリックデバッガSYMDEBの実行例

シンボリックデバッガが威力を発揮するのは、とくに高級言語とアセンブラルーチンをリンクしたときです。普通は、高級言語にアセンブラのルーチンをリンクすると、ブレークポイントをセットしようにも、そのアドレスが簡単にはわかりません。それがソースファイルのシンボル名で指定できるのですから、その威力が想像できると思います。

■ ソースコードデバッガ

MASMのバージョン5.1やMS-Cのバージョン4.0以降に付属しているソースコードデバッガCODEVIEWでは、ソースコードと実行プログラムを対比させながらデバッグを行うことができます。

CODEVIEWは、C言語などの高級言語のデバッグ用に開発されたもので、従来ならばマシン語オブジェクトを逆アセンブルしながら行っていたデバッグ方法に対し、ソースコード上でブレークポイントをかけたり、変数の値を確認したりできる、画期的なデバッガです。さらに、ウィンドウやメニューを持つなど、ユーザーインターフェイスが格段に改良されていることも特徴です。

なお、CODEVIEWについては、6章で改めて解説します。

5.7　ライブラリの活用

以上で、アセンブラによるソフトウェア開発の流れは、ひととおり理解できたのではないかと思います。最後に、モジュール別プログラミングの最大の利点である、ライブラリの作成について解説しましょう。

ライブラリとは、いろいろな開発において作成した多くのモジュールの中で、あとで再利用できそうなものをまとめて整理しておくものです。本章の例題プログラム（16進電卓プログラム）でいえば、16進ASCII文字列とバイナリ数値との変換ルーチンなどは利用価値が高く、これからも各種のプログラムにおいて、値を表示するなどに利用できるでしょう。

いろいろなアプリケーションプログラムをいつくも作っていくうちには、このようなモジュールがたくさん集まります。新しいプログラムを作成する際には、それらのモジュールの中のいくつかは、きっと利用できるはずです。

このようなモジュールを再利用するには、各モジュールをOBJファイルの形で保存しておき、リンク時にそのファイル名を指定してリンクする手もあります。しかし、モジュールの数が多くなると、その管理が非常にたいへんになり、この方法は現実的ではなくなります。そこでこれらのモジュールを1つのファイルに格納し、まとめて管理できるようにしたものがライブラリです。

前述のように、OBJファイルのままでは、その管理もたいへんであり、多くのモジュールを利用す

るときには、リンク実行時のコマンドラインに、そのすべてを指定しなければなりません。これもたいへんな作業です。

これをライブラリにすると、1つのファイルを管理すればよいことになり、取り扱いが便利になります。リンク時にライブラリファイルを指定することにより、ソースプログラム中で呼んでいる(EXTRN 指定をした上で呼ぶ)モジュールを、そのライブラリファイルの中からリンカが自動的に検索し、リンクしてくれるのです。とくに高級言語などは、このライブラリの機能がなければ、使いものにならないといってもよいでしょう。

MS-DOS では、ライブラリを管理するためのプログラムであるライブラリマネージャLIBが用意されています。LIB は、ライブラリの作成、モジュールの登録、抜き出し、削除、リスティングなどを行う機能を持っています。

LIB を起動すると、メッセージに続けて次のようにライブラリファイルの名前を問い合わせてきます。

(1) Library name：
ここでは更新するライブラリの名前を指定する。ファイルタイプを省略すると LIB ファイルとみなされる。

(2) Library does not exist. Create?
(1)で指定したライブラリファイルが存在しない場合に、表示される。新しいライブラリを作成するときには「Y」(y)と答える。

(3) Operations：
ここでは指定したライブラリに対する操作を指示する。使用できる LIB のコマンドについては後述する。

(4) List file：
任意のファイル名を指定することにより、ライブラリのクロスリファレンスファイルを作ることができる。必要なければ単にリターンを入力する。

(5) Output library：
更新されたライブラリを、指定したその任意のファイル名で出力する。単にリターンキーを入力すると、もとのライブラリ名が付けられる(もとのライブラリが更新される)。

コマンド	機能	
＋	追加	ライブラリにモジュールを追加する
−	削除	ライブラリからモジュールを削除する
−＋	置換	ライブラリ内のモジュールを新しいモジュールと置き換える
＊	抽出	ライブラリからモジュールを抽出する
−＊	移動	ライブラリからモジュールを抜き取る

表 5.4　ライブラリ操作コマンド

以上は、LIB を対話形式で実行した例ですが、これらのパラメータをコマンドラインで一度に指定することもできます。

ライブラリに対する操作コマンドを表 5.4 にまとめておきましょう。また、これらのコマンドを使った LIB の実行例を図 5.27 に示します。

```
A>LIB↵

Microsoft Library Manager
Version 3.00 (C)Copyright Microsoft Corp 1983, 1984, 1985

Library name: MYLIB↵
Library does not exist.  Create? Y↵
Operations: +OPE↵
List file:↵

A>DIR *.LIB↵

 ドライブ A: のディスクのボリュームラベルは APP MS-DOS
 ディレクトリは A:¥WORK¥ASM¥CALC

MYLIB    LIB     1024  89-09-22    1:35
        1 個のファイルがあります.
   308224 バイトが使用可能です.

A>LIB MYLIB +HEX *OPE;↵

Microsoft Library Manager
Version 3.00 (C)Copyright Microsoft Corp 1983, 1984, 1985

A>
```

A>LIB↵ ……ライブラリマネージャを起動する

Library name: MYLIB↵ ……ライブラリのファイル名を入力する

Create? Y↵ ……新しく作成するファイルなので「Y」を入力する

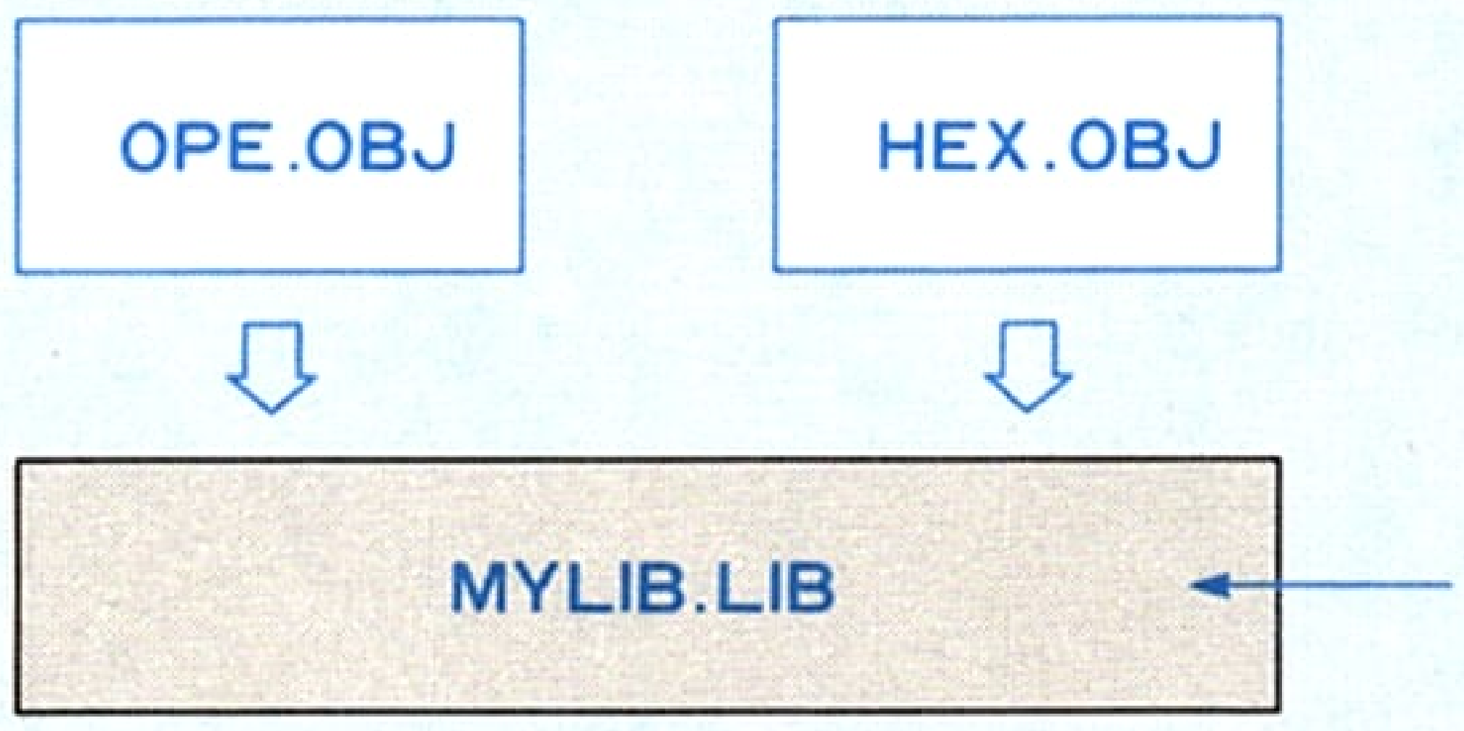
Operations: +OPE↵ ……オブジェクトファイル「OPE.OBJ」をライブラリ「MYLIB」に加える

List file:↵ ……クロスリファレンス・リスティングファイル名を入力する
必要のないときはリターンキーのみを入力する

A>DIR *.LIB↵ ……作成されたライブラリの確認

MYLIB LIB 1024 89-09-22 1:35 ……新しいライブラリが作成されている

A>LIB MYLIB +HEX *OPE;↵ ……今回はコマンドラインで実行する。上で作成したライブラリ「MYLIB」に、オブジェクトファイル「HEX.OBJ」を追加し、ライブラリ内のモジュール「OPE」のコピーを「OPE.OBJ」として作成する（ここでの実行例では、すでにこのファイルが存在しているが）



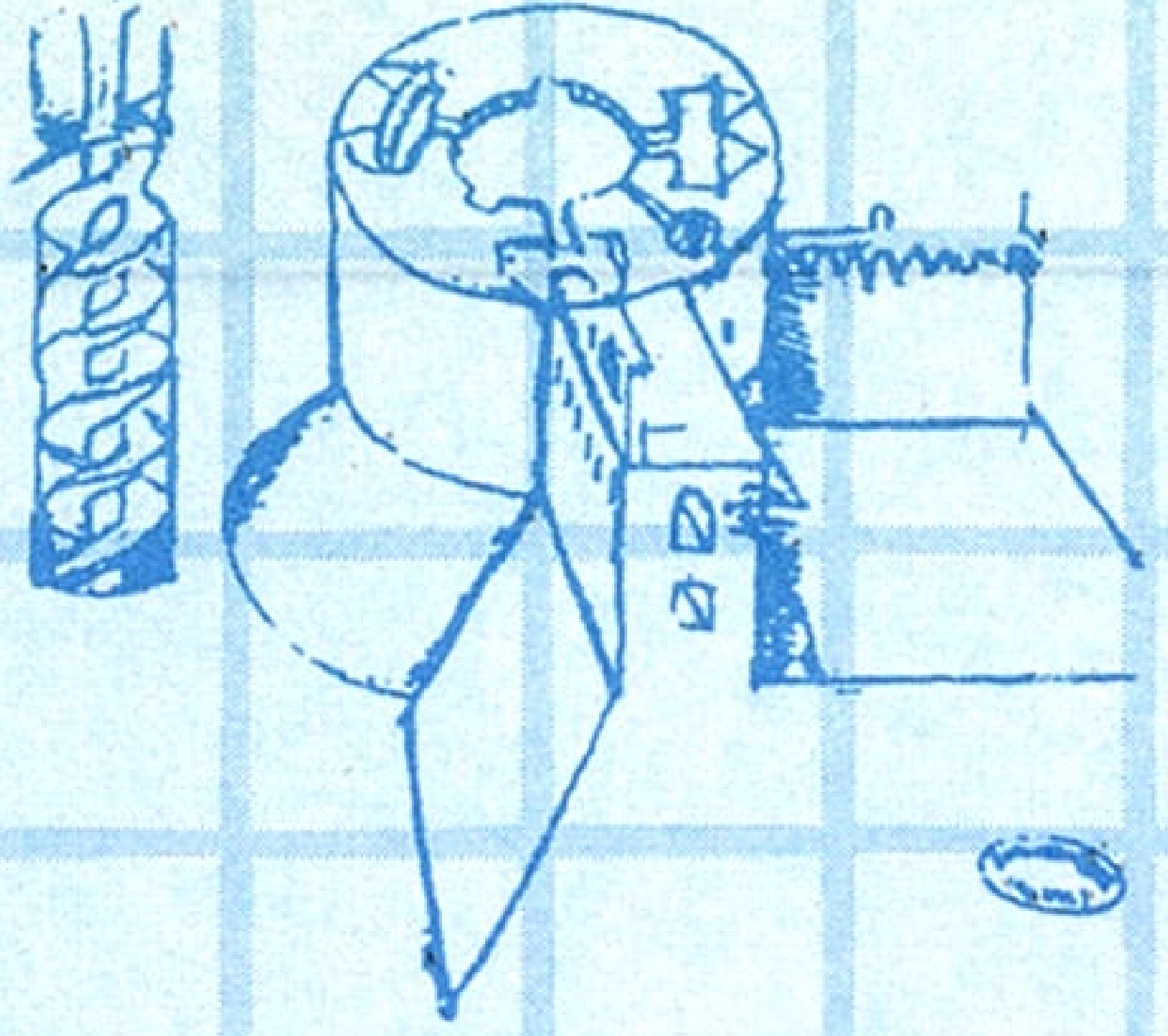
以上の操作で 2 つのモジュールを含んだ
ライブラリファイルができた

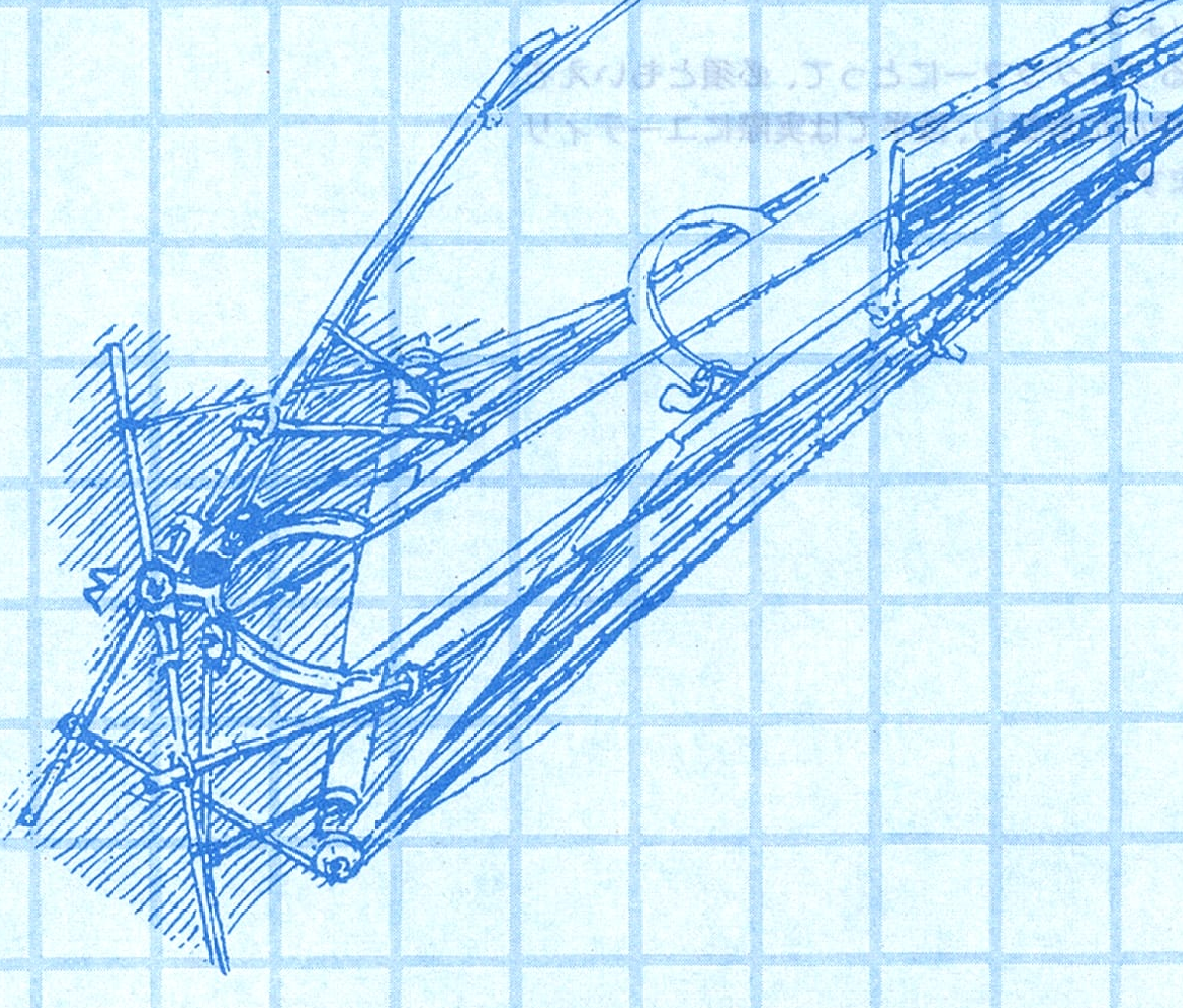
図 5.27　ライブラリマネージャLIB によるライブラリファイルの作成

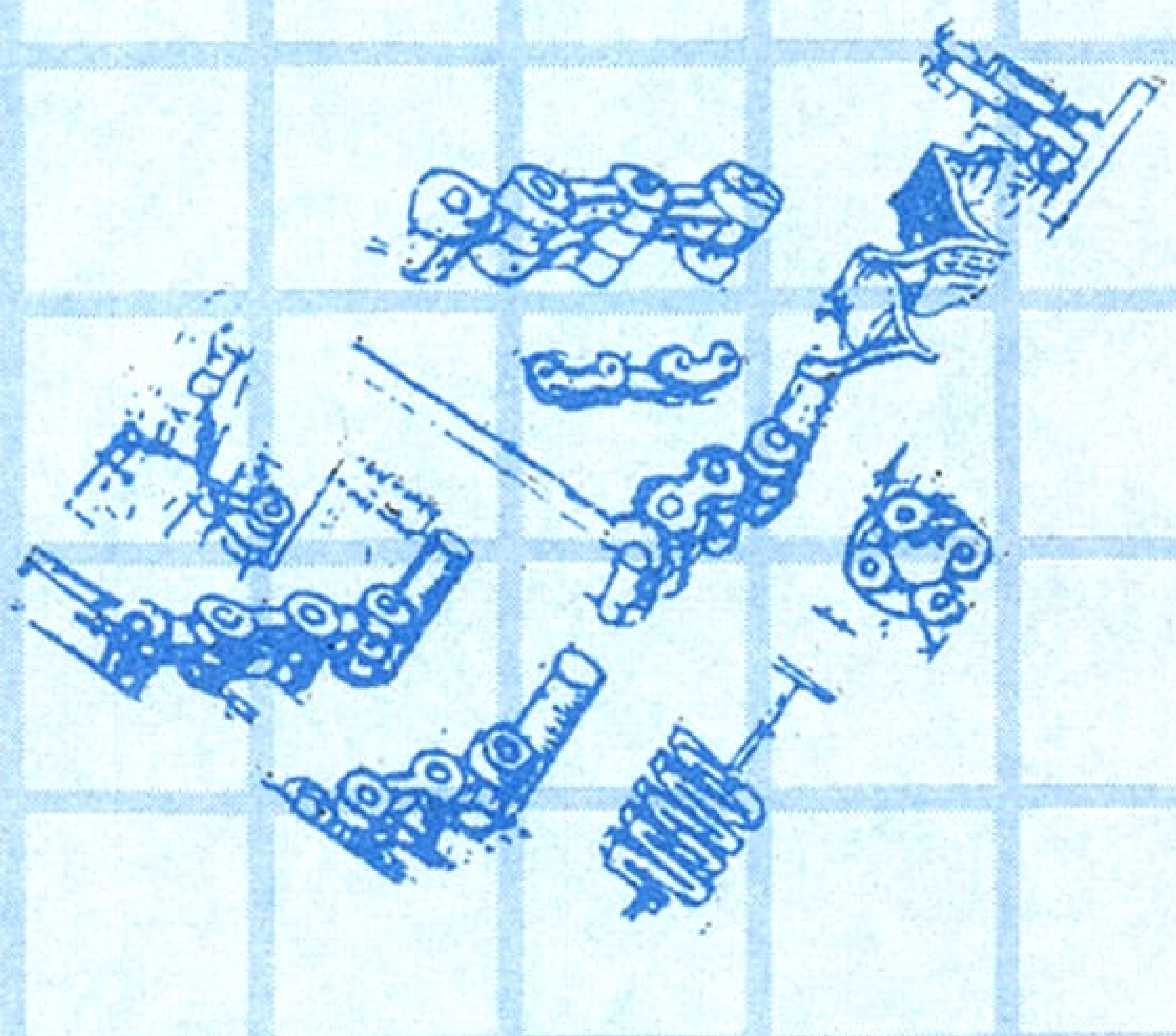
いくつものプログラムを作成するうちには、自分のライブラリがどんどん増えていくものです。ライブラリはプログラマーの財産ですし、使いやすいライブラリを作れば、他人が利用することもできます。また、高級言語には、多くの各種の機能を持ったライブラリが付属しているのが普通です（6 章参照）。

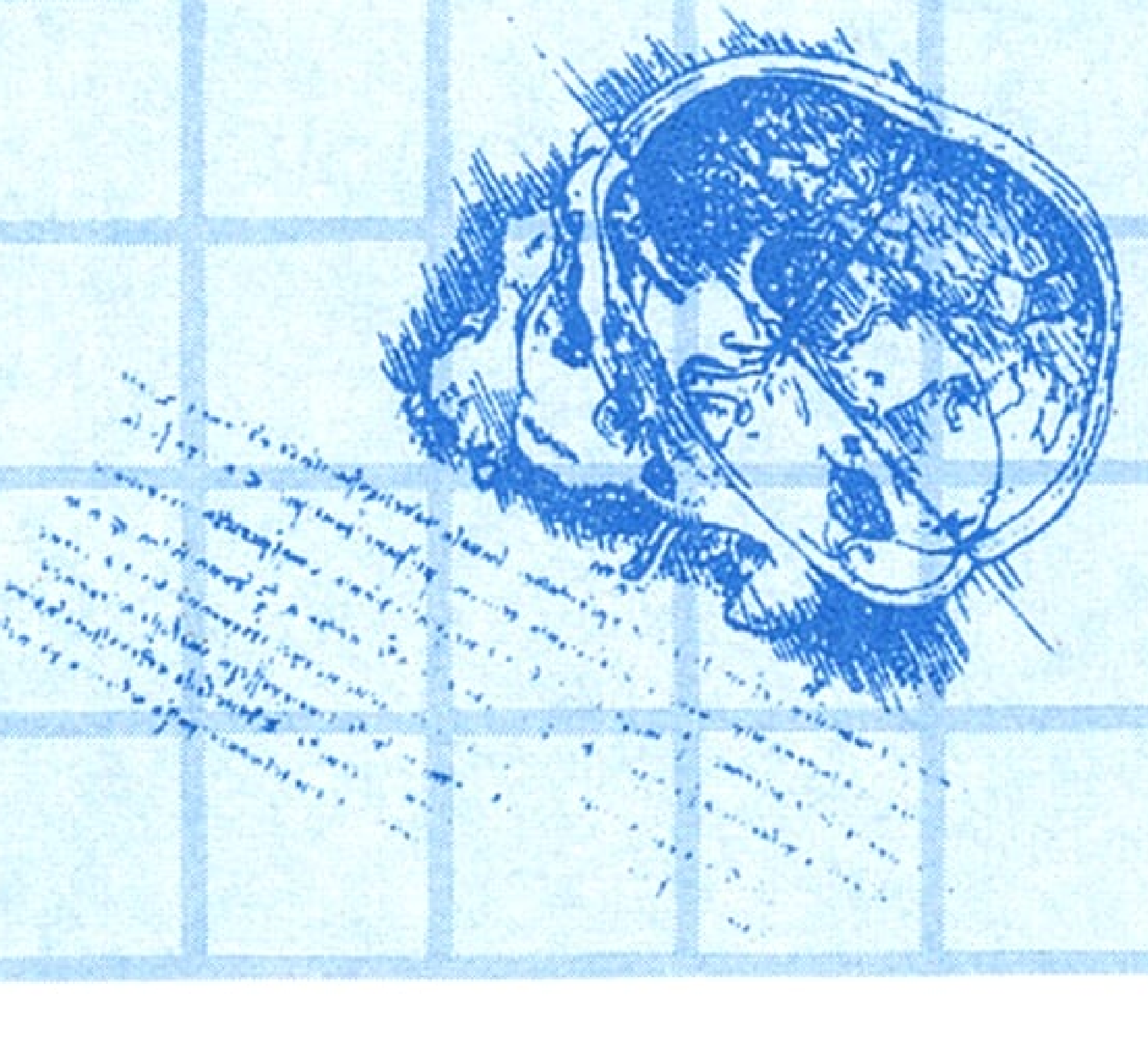
みなさんもプログラムを作成するときには、モジュラ・プログラミングを強く意識して、優れたライブラリを作るように心掛けてください。ライブラリこそ、ソフトウェア的に最も高価な財産なのです。

6章
C言語によるプログラム開発









筆者は実際の開発を通じて、MS-DOS上で大きなソフトウェア開発を行う場合の、現在最も実用的なプログラミング言語はC言語であるという確信を持っています。C言語は、UNIXを記述しているプログラミング言語として広く知られていますが、そのことがUNIXの評価とともに、C言語を今日のようにポピュラーなものにしたといってもよいでしょう。C言語とUNIXは切り離して考えることはできません。CはUNIX、UNIXはCなのです。

MS-DOSは、バージョン2.*x*でUNIXの思想を大幅に取り入れましたが、MS-DOSがUNIXに近づいただけ、C言語との関係も深まっています。C言語は今やMS-DOSにとって、なくてはならないプログラミング言語の1つです。「自分でプログラムを作ろう」「MS-DOSをソフトウェア開発に利用しよう」と考えている方は、いろいろな面からみて、現時点ではC言語を使うことが最も有利でしょう。

本章では、MS-DOSを利用するプログラマーにとって、必須ともいえるC言語の基礎知識を、その環境を中心に解説し、後半では実際にユーティリティプログラムの作成を実習します。

6.1　C言語の世界

MS-DOSはバージョン2.*x*からUNIXの思想を大幅に取り入れましたが、このUNIXを記述したプログラミング言語がCであることはご存知の方も多いでしょう。そしてUNIXの成功が、C言語を今日ほど広く普及させ、その機能や環境を育ててきた理由にもなっています。

UNIXにおいては、OSとのインターフェイスのすべては、C言語のライブラリ関数として提供されています。また、C言語はシステムの記述に向いたプログラミング言語ともいわれていますが、UNIXのカーネルやほとんどのコマンド群、ツール群はC言語で書かれています。

MS-DOS上のC言語にも、MS-DOSとのインターフェイスであるシステムコールの多くが、ライブラリ関数として提供されています。また、MS-DOSのシステムコールの多くはUNIXのものに近く、MS-DOSのコマンドやユーティリティプログラムを新しく開発するにも、やはりC言語が最適でしょう。C言語はMS-DOSと非常に相性のよいプログラミング言語なのです。

■C言語はアセンブラだ

『C言語は高級言語であり、アセンブラとは基本的な違いがある。でも、アセンブラと思えばそんな気もする。そうだ、きっとアセンブラに違いない！』――C言語を長く使っていると、そう思うようになってきます。実際、ビット演算やシフト演算ができるという点から、C言語はアセンブラに近い低級言語(CPUに近いという意味)であるという人もいます。まあ、そのような議論はC言語の専門書に任せておいて、ここではソフトウェアの開発環境としてのC言語の特徴を捉えていきましょう。

C言語では、マシン語レベルで(つまりアセンブラで)書かないと不可能に思えるようなシステムとのやりとりなども簡単に書くことができ、そのような細かい処理ができるという点で、アセンブラに近いプログラミング言語であるともいえます。5章でアセンブラのマクロ機能について触れましたが、そのマクロ機能を徹底的に追求し、CPUを見えないまでに発展させてしまったのがC言語であると考えることもできます。

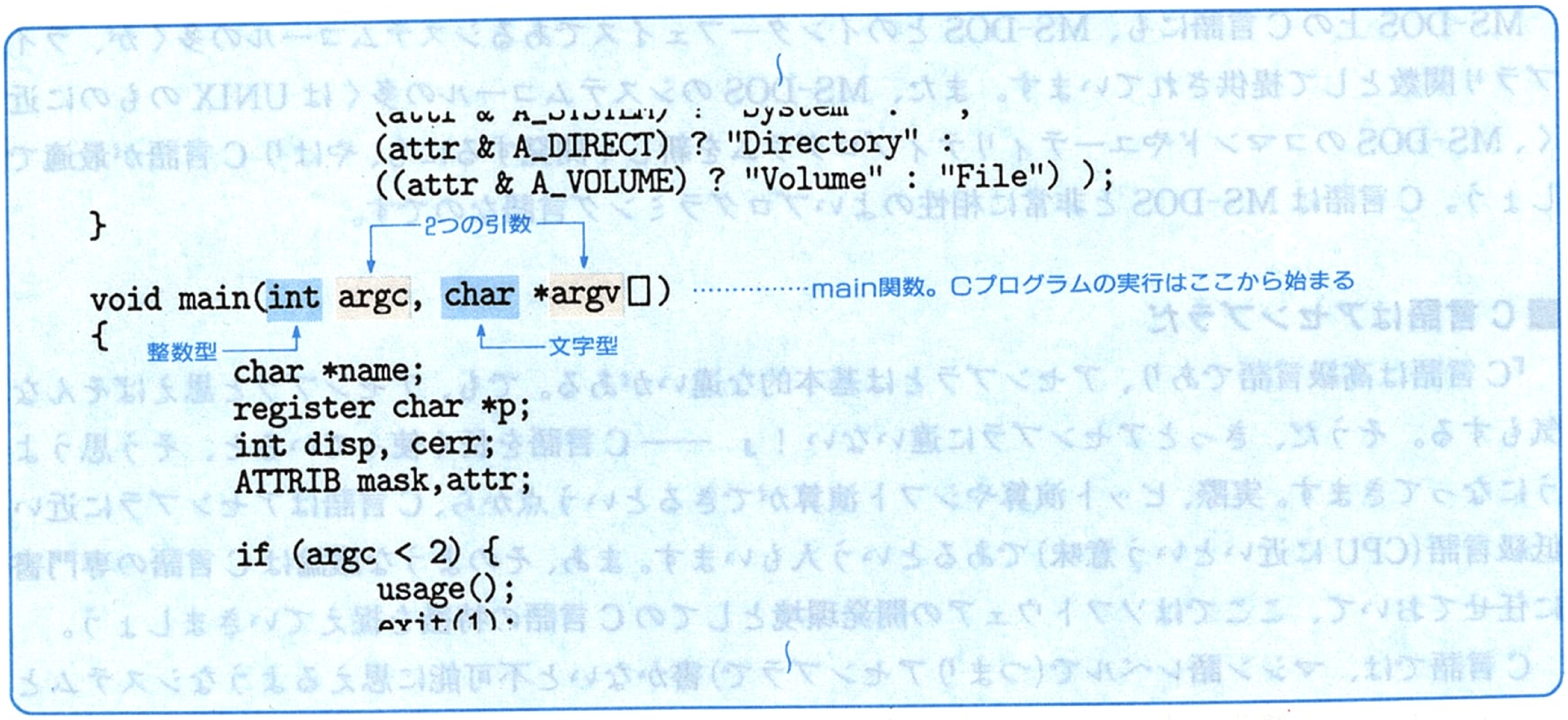
C言語は、アセンブラ並みの小回りがきき、しかもアセンブラよりもはるかに快適な開発環境を提供してくれます。しかしこれは、C言語自身の文法などのプログラミング言語仕様だけの特徴ではなく、C言語を取り巻く環境、その中でもとくに、豊富で強力なライブラリ群にささえられているおかげです。

ここでC言語がどれほど便利な「アセンブラ」であるかを紹介するために、本章の6.4節で作成する例題プログラムを中心に、ほんの少しだけC言語の世界をのぞいてみましょう。

■ C プログラムの入り口と出口

まず、本章 6.4 節のリスト 6.1 に示すファイル属性変更プログラム chmod（1 章で作成した CHMOD の機能を拡張した C 言語版）を例に、プログラムの入口と出口（実行開始点と終了点）から見ていきましょう。その部分を**図 6.1** に引用しておきます。

C 言語のプログラムは main という関数から実行が開始されますが、この関数には引数が 2 つあり、そこにコマンドラインに関する情報が収められています。完成した実行可能なオブジェクトファイルには、プログラムをスタートさせるためのスタートアップルーチンが C 言語によって組み込まれていますが、このルーチンはその 2 つの値をセットしてから main 関数を呼び出します。



```
                (attr & A_DIRECT) ? "Directory" :
                ((attr & A_VOLUME) ? "Volume" : "File") );
}

void main(int argc, char *argv[])
{
        char *name;
        register char *p;
        int disp, cerr;
        ATTRIB mask,attr;

        if (argc < 2) {
                usage();
                exit(1)
```

図 6.1　C プログラムの実行開始の処理

引数 argc には、コマンド名を含めたコマンドライン上のパラメータの個数がはいります。そして argv には何と、コマンドラインが各パラメータに分解処理され、パラメータ文字列となったものがはいっているのです。3 章や 4 章でも触れたように、PSP のオフセット 0080H からは、コマンドラインの文字列がそのまま連続した文字列として格納されています。アセンブラでこのパラメータを取り出すようなプログラムを書こうとすると、PSP のコマンドライン文字列を参照して、各パラメータの区切りを識別するだけでも、実にたいへんな仕事です。1 章のプログラムで、パラメータの解析にかなりの部分をさいていることでもわかるでしょう（1 章のリスト 1.1 参照）。C 言語ではこれを 1 つひとつのパラメータに分解して渡してくれるのです。たとえば、**図 6.2** のような処理が行われます。

ただし、MS-DOS のバージョン 2.*x* では、MS-DOS の機能上、メインプログラムのコマンド名（プログラムファイル名。図 6.2 の例では「COM」）は、C 言語側に渡せないのでセットされません。しかしこの問題は、MS-DOS バージョン 3.*x* では解決され、コマンド名も渡されるようになっています。

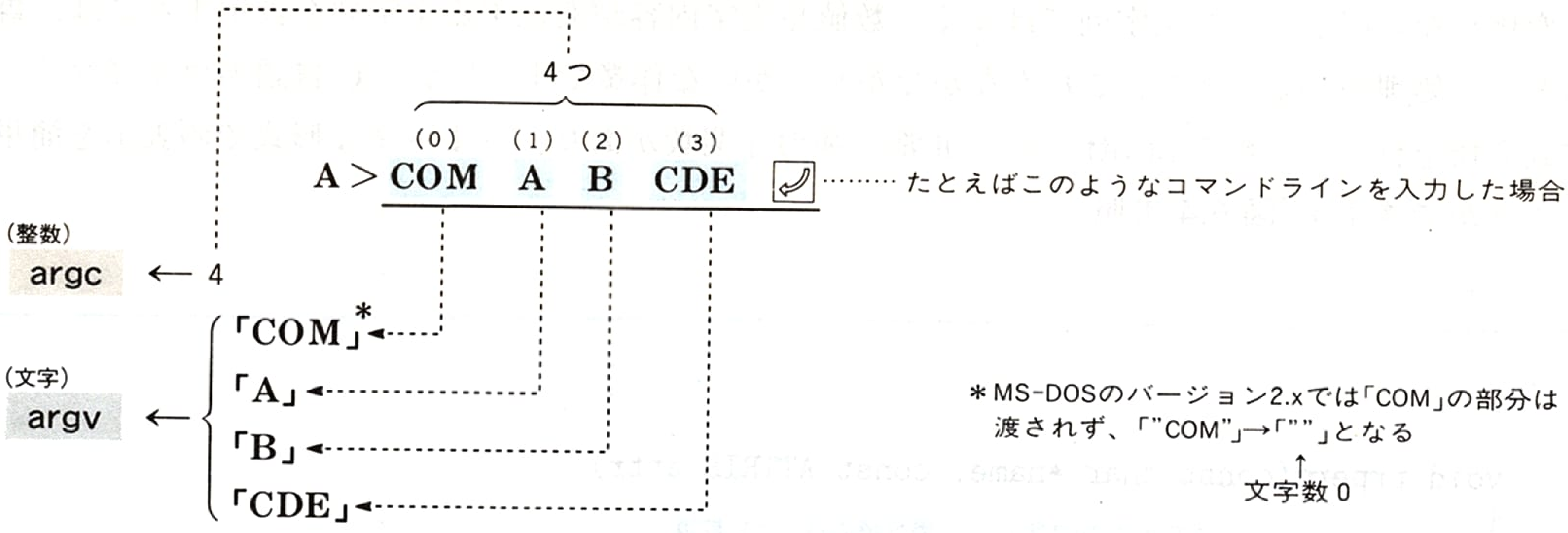

4 つ
(0) (1) (2) (3)
A＞COM A B CDE ↵
……たとえばこのようなコマンドラインを入力した場合
(整数)
argc ← 4
(文字)
argv ←
「COM」*
「A」
「B」
「CDE」
＊MS-DOSのバージョン2.xでは「COM」の部分は渡されず、「"COM"」→「""」となる
文字数 0

図 6.2　C 言語によるコマンドライン文字列の処理

次はプログラムの終了です。C 言語のプログラムを終了するには exit という関数を使います。main 関数の終わり（最後の「}」）に達した場合にも、最終的には exit 関数が自動的に呼ばれることになりますが、この関数に引数を渡すことにより、親プロセスに制御権を渡し、リターンコードを返すことができます（図 6.3 参照）。

```
}

void main(int argc, char *argv[])
{
        char *name;
        register char *p;
        int disp, cerr;
        ATTRIB mask,attr;

        if (argc < 2) {
                usage();
                exit(1);
        }

        while (--argc > 0) {
```

前述のプログラムの始まり

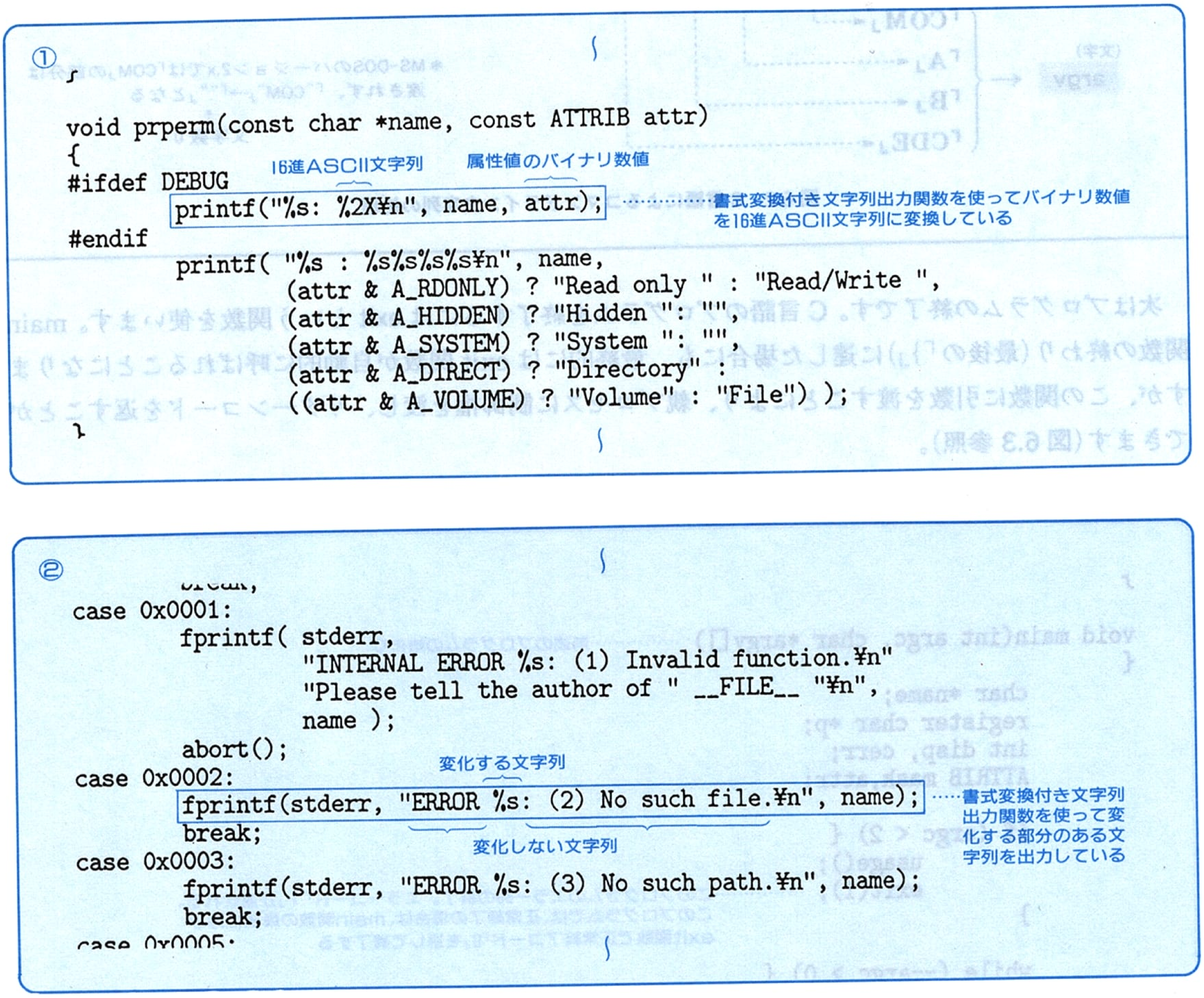
このプログラムのエラー時の終了。エラーコード「1」が返される。このプログラムでは、正常終了の場合は、main関数の最後に呼ぶexit関数で正常終了コード「0」を返して終了する

図 6.3　C プログラムの終了処理

このリターンコードは、ファンクションリクエストの 4C$_H$ で返すリターンコードとまったく同じもので、バッチコマンド中で使用可能な ERRORLEVEL の値になります。この値を参照することにより、プログラムの外からそのプログラムがどういう状態で終了したかを調べることが可能です。

■ 書式変換

変化しない固定された文字列ではなく、数値や文字内容が変化する文字列を表示するには、書式変換という処理が必要であり、これもなかなかやっかいな作業です。しかしC言語のライブラリには、書式変換を行ってくれるprintfという非常に強力な関数があり、さまざまな形式での表示を簡単に行うことができます(図6.4参照)。



①

```
}

void prperm(const char *name, const ATTRIB attr)
{
#ifdef DEBUG
        printf("%s: %2X¥n", name, attr);
#endif
        printf( "%s : %s%s%s%s¥n", name,
                (attr & A_RDONLY) ? "Read only " : "Read/Write ",
                (attr & A_HIDDEN) ? "Hidden": "",
                (attr & A_SYSTEM) ? "System ": "",
                (attr & A_DIRECT) ? "Directory" :
                ((attr & A_VOLUME) ? "Volume": "File") );
}
```

16進ASCII文字列

属性値のバイナリ数値

書式変換付き文字列出力関数を使ってバイナリ数値を16進ASCII文字列に変換している

②

```
break;
case 0x0001:
        fprintf( stderr,
                "INTERNAL ERROR %s: (1) Invalid function.¥n"
                "Please tell the author of " __FILE__ "¥n",
                name );
        abort();
case 0x0002:
        fprintf(stderr, "ERROR %s: (2) No such file.¥n", name);
        break;
case 0x0003:
        fprintf(stderr, "ERROR %s: (3) No such path.¥n", name);
        break;
case 0x0005:
```

変化する文字列

変化しない文字列

書式変換付き文字列出力関数を使って変化する部分のある文字列を出力している

図6.4　printf、fprintf関数による書式変換

図6.4①に示したのは、リスト6.1に示すプログラム「chmod.c」内の、属性値を16進2桁のASCII文字列で表示するための部分です。アセンブラでこれを行う場合は、5章で作成したような、バイナリ数値から16進ASCII文字列への変換プログラムを、何行にもわたって書かなければなりません(5章のリスト5.4参照)。

また、図6.4②に示したのは、変化するファイル名を表す文字列とともに、変化しない、メッセージを表示する部分です。この部分をアセンブラで書けば、5章のリストでもわかるように、変化する部分とそうでない部分とを分けて、別々の表示ルーチンを作るしかないでしょう。

■ プログラム実行中のレジスタ内容の表示

プログラムのデバッグ時に、プログラムを実行させながら目的のレジスタや変数の値を表示して観察したい場合があります。アセンブラでは、その表示処理の部分をプログラムに組み込むのは困難ですが、C言語ではさきのprintfを1行書くことで実現できます。しかも、それを#ifdef〜#endifという命令で囲んでおけば、デバッグ終了後には、コンパイル時のオプション、あるいはプログラム内の#define行を変更することにより、その部分を取り除くことができます。コンパイルの仕方や宣言文の変更だけで、プログラムの本体には手を加えることなく、その部分を取り除くことができるのです(図6.5参照)。

```
ʃ

void prperm(const char *name, const ATTRIB attr)
{
#ifdef DEBUG
        printf("%s: %2X¥n", name, attr);
#endif
        printf( "%s : %s%s%s%s¥n", name,
                (attr & A_RDONLY) ? "Read only " : "Read/Write ",
                (attr & A_HIDDEN) ? "Hidden ": "",
                (attr & A_SYSTEM) ? "System ": "",
                (attr & A_DIRECT) ? "Directory" :
                ((attr & A_VOLUME) ? "Volume" : "File") );
}
ʃ
```

得られた変数(この場合は属性の値)をそのつど表示するデバッグ用の部分

図6.5　デバッグのための変数の表示例(「chmod.c」の場合)

■ システムコールの直接呼び出し

たとえば、さきに述べた文字列を表示するライブラリ関数などは、その内部ではMS-DOSの表示に関するシステムコールが使われていることは容易に想像できます。C言語には、このようなライブラリ関数内部でシステムコールが使われるだけでなく、システムコールそのものを直接実行する機能があります。つまり、CPUの各レジスタに目的のファンクションをコールするための値をセットして、INT 21Hを実行する関数が用意されているのです。この機能は、MS-DOS上のCコンパイラのほとんどに用意されています。

図6.6に示すのは、本章のchmodプログラムの3つのソースファイルのうち、リスト6.2の「attrib.c」に見られる、MS-DOSシステムを直接コールしている部分です。

```
/*
 * change file attributes
 */
USHORT attrib(const char *path, const ATTRIB mask, PATTRIB attr)
{
        union REGS reg;

        reg.x.ax = 0x4300;
        reg.x.dx = (USHORT)path;
        intdos(&reg, &reg);
        if (reg.x.cflag & 1) {
                doserrno = reg.x.ax;
                return doserrno;
        }
        *attr = ((reg.x.cx & ~mask) | (mask & *attr));
        if (*attr != reg.x.cx) {
                reg.x.ax = 0x4301;
                reg.x.cx = *attr;
                intdos(&reg, &reg);
                if (reg.x.cflag & 1) {
                        doserrno = reg.x.ax;
                        return doserrno;
                }
```

指定したファイルの属性を得るためのファンクションリクエストの部分

ファンクションリクエストのファンクションナンバー（AHレジスタ）

属性を「得る」ためのパラメータ（ALレジスタ）

パス名（ファイル名）のアドレス（DXレジスタ）

上でセットした値をここで各レジスタにセットし、INT 21Hを実行する。そして実行後の各レジスタの値が返される

フラグレジスタの値が返されている

指定されたファイルの属性を得るファンクションリクエストの部分

キャリーフラグが「1」でなければCXレジスタに得られた属性が返されている

注）このプログラムはスモールモデルでしか動作しません。ラージモデルではDSレジスタも設定する必要があります。

図6.6　システムコールを直接操作する例

この機能を使えば、MS-DOSのすべてのシステムコールを直接利用することができます。アセンブラを使えば、当然のことながらMS-DOSのシステムを余すことなく利用できますが、C言語でもまったく同様にすべてのシステムコールを利用できるのです。この機能があればほとんどの場合、C言語はアセンブラに取って代わることができるでしょう。

図6.6に示した部分は、C言語からシステムコールを行う典型的な例題です。これを参考にすれば、C言語からMS-DOSシステムの任意のファンクションを自由に呼び出し、処理することができるでしょう。

■ ライブラリあってのC言語

ここまで読み進んできたみなさんは、あるいはお気付きかもしれませんが、C言語の使いやすさは、その多くがライブラリ関数が提供されていることによるものです。5章のアセンブラに関する解説では、そのマクロ機能を使いこなし、汎用性のあるモジュールをたくさん集めたライブラリを用意して

おけば、かなり効率のよいプログラムミングができることをお話ししました。ところが、C言語にはそのような機能やライブラリが最初から用意されています。

このように、C言語自体はアセンブラと考えてプログラミングすることができます。C言語の各処理系（各社の製品）による差は、文法自体に関しては基本的にはないといってもよいでしょう。しかし、処理系に付属するライブラリは各社が独自に提供しているものであり、市販されている中から、ある処理系を選ぶというのは、それに付属しているライブラリを選ぶことでもあります。この選択による内容の差は開発環境に大きく影響することにもなるでしょう。

■C言語の各処理系間の互換性

C言語では、入出力を含むすべてのライブラリ関数は、言語仕様には含まれていません。しかし、UNIXで用意されている関数を基本としているため、ほとんどの処理系で、かなりの互換性が保たれており、ゆえにC言語は、「CPUを選ばないアセンブラ」と呼ばれたりもします。アセンブラで書かれたプログラムをほかのCPUに移植するには、プログラムのほとんどすべてを書き直すしかありません。これがC言語であれば、書き直す部分はかなり限られてきます。

ただし、ここで用いた「ほとんど」とか「かなり」という言葉は実は曲者であり、細かい部分では多くの処理系間の互換性はありません。互換性が保たれているのは、UNIX上のC言語のライブラリをもとにした、いわゆる標準関数と呼ばれるものだけです。とくに、C言語の事実上のプログラミング言語仕様書ともいうべき『The C programming language』[*1]に書かれた仕様やANSI-C[*2]と呼ばれるC言語の標準規格は守られているようですが、それ以外のライブラリ、つまりMS-DOSの機能を直接呼び出す関数やグラフィックスを扱う関数など、各処理系で独自に拡張したライブラリの互換性は、あまり保証されていないのが現状です。

■C言語のエラーチェック

C言語は、高級言語でありながら、プログラムのミスに関してはアセンブラに近いといってよいほど、エラーのほとんどがチェックされません。アセンブラのように低級言語では、プログラムのミスによって簡単に暴走し、リセットボタンを押さなければならないはめに陥りますが、C言語も同じです。それというのも、コンパイル時に必要最低限のチェックを行うだけで、実行時にも、高級言語で行われているような配列の上限などのチェックを、まったくといっていいほど無視しているからです。C言語は、アセンブラのプログラムと同様に、そのようなチェックをすべてプログラマー自身が行わなければなりません。その意味でもC言語はアセンブラに近いといえます。

[*1] 『The C programming language　Second　Edition』　B.W.カーニハン・D.M.リッチー著。
（邦訳）『プログラミング言語C　第2版』　石田晴久訳　共立出版刊　1989（ただしこの本は初心者向きではない）

[*2] ANSIとは、American　National　Standards　Institute（米国規格協会）のこと。

またプログラムのミスに関して、以前は**LINT**というコマンドを使ってソースファイルの正しさをチェックしていましたが、最近では引数の型を**プロトタイプ宣言**としてあらかじめ宣言しておくことで、コンパイル時にチェックする処理系が主流になっています。

6.2　C言語とMS-DOSとの相性

前節でも触れましたが、さらにいくつかの例を示してC言語とMS-DOSとの相性のよさを証明してみましょう。

■ ファイルのオープン／クローズ、読み出し／書き込み

2章で解説した、ファイルのオープン／クローズや、読み出し／書き込みのシステムコールは、C言語のライブラリ関数にそっくりそのまま用意されています(図6.7)。

int open (const char *name, int mode, int pmode) ファイル名　オープンモード　アクセスモード	指定した名前のファイルを指定したモードでオープンし、ファイルハンドルを返す。ただし、処理系によっては、アクセスモードを指定できないものもある
int open (const char *name, int mode)	
int creat (const char *name, int pmode)	指定した名前のファイルを指定したアクセスモードで作成し、ファイルハンドルを返す
int read (int fd, char *buf, int n) ファイルハンドル　アドレス　サイズ	ファイルハンドルで指定したファイルから、指定したバッファへ指定したバイト数書き込む
int write (int fd, char *buf, unsigned int n)	ファイルハンドルで指定したファイルへ、指定したバッファから指定したバイト数書き込む
int close (int fd)	ファイルハンドルで指定したファイルをクローズする

図6.7　ファイル操作に関するライブラリ関数の例

これはファイルハンドルを使ったファイルアクセスのシステムコールそのものです。各関数に渡すパラメータは、システムコールのときにレジスタにセットする値であり、またopenおよびcreat関数が返す値は、2章で解説したファイルハンドルの値にほかなりません。

■ バッファリング

C 言語でも、システムコールを使うことによって任意の長さのデータを読み書きすることができますが、1 バイトや 2 バイト単位で読み書きする場合、そのたびにシステムコールを使っていたのでは、むだな処理が多くなり、実行が遅くなります。このため C 言語では、独自のバッファリング(MS-DOS によるバッファリングとは別の)を行うライブラリ関数が用意されています (図 6.8)。

FILE *fopen (const char *$\underset{\uparrow\ \text{ファイル名}}{\underline{\text{name}}}$, const char *$\underset{\uparrow\ \text{オープンモード}}{\underline{\text{mode}}}$)	指定した名前のファイルを指定したモードでオープンし、ファイルポインタを返す
int getc (FILE *$\underset{\uparrow\ \text{ファイルポインタ}}{\underline{\text{fp}}}$)	ファイルポインタで指定したファイルから 1 文字読み込む
int putc (int c, FILE *fp)	ファイルポインタで指定したファイルへ 1 文字書き込む
int fclose (FILE *fp)	ファイルポインタで指定したファイルをクローズする
int fprintf (FILE *fp, const char *format,…)	ファイルポインタで指定したファイルへの書式付き書き込み (printf( ))
int fscanf (FILE *fp, const char *format,…)	ファイルポインタで指定したファイルからの書式付き読み込み(scanf( ))

図 6.8　バッファリングを行うライブラリ関数の例

基本的には、これらの関数を使うことで、細かなファイルアクセスが可能です。これらの機能は、前項のファイルハンドルによる入出力と組み合わせることによって実現されています。

たとえば、のちほど作成するサンプルプログラム chmod に使われている関数 fprintf を使えば、ファイルに書式付きの書き込みができます。この部分のリストを抜き出して図 6.9 に示しましょう。またこの例にあるように、標準エラー出力に出力することも簡単にできます。つまり、リダイレクトされていてもエラーメッセージをディスプレイに表示することができるのです。これはほかのプログラミング言語にはあまり見られない機能でしょう。このバッファリングによる入出力に関しては、さらに豊富な関数を用意している処理系もあります。

```
break;
case 0x0001:
        fprintf( stderr,
                 "INTERNAL ERROR %s: (1) Invalid function.¥n"
                 "Please tell the author of " __FILE__ "¥n",
                 name );
        abort();
case 0x0002:
        fprintf(stderr, "ERROR %s: (2) No such file.¥n", name);
        break;
case 0x0003:
        fprintf(stderr, "ERROR %s: (3) No such path.¥n", name);
        break;
case 0x0005:
        fprintf(stderr, "ERROR %s: (5) Access denied.¥n", name);
        break;
```

これは、さきの図6.4と同じ部分です

この関数によって出力されるメッセージは、リダイレクトの影響を受けず、常にディスプレイに表示される

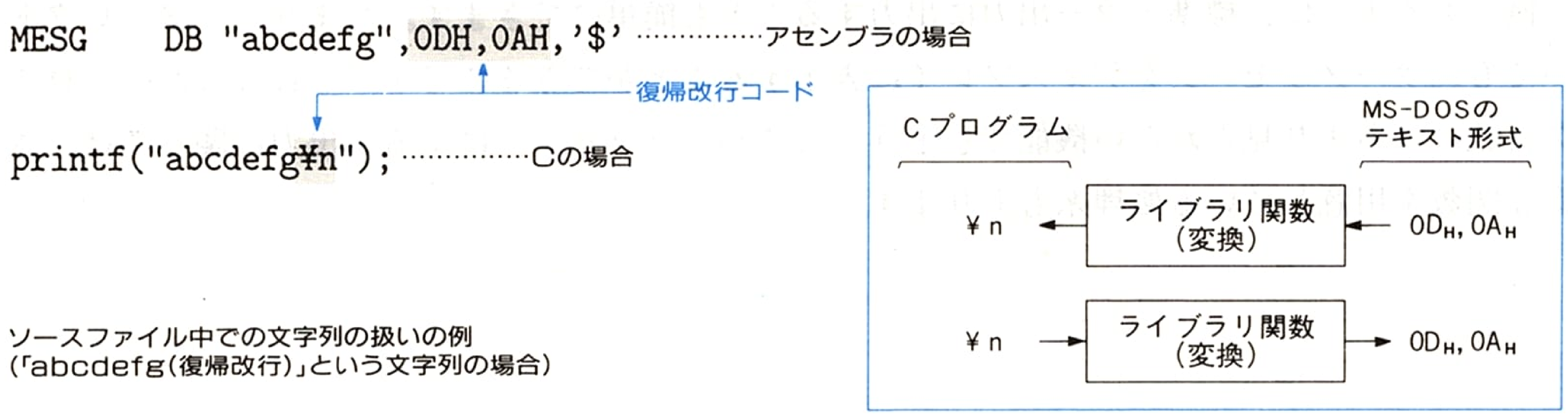
標準エラー出力へ出力する

図6.9　バッファリングによる入出力の関数例

■ 復帰改行コードの相違についての注意

さきに示したバッファリングによる入出力は、互換性に関して注意しなければならないことがあります。C言語の復帰改行コード「¥n」の問題は、MS-DOSでは復帰改行コードを「CR」「LF」の2文字(0DH、0AH)として扱っているのに対し、UNIXでは(つまりC言語でも)1文字の「¥n」で扱っているために生じます。この問題は、ほとんどのC言語でバッファリングによる入出力関数により自動的に変換、吸収されます。ただし、吸収はされますが、この復帰改行コードに関しては、C言語とMS-DOSのテキスト形式との相性が悪いということになりますので注意してください(図6.10参照)。

なお、バイナリファイルを扱うときには、「¥n」→「CR, LF」の変換が行われては困るので、通常の処理系では、「変換する／しない」を指定できるようになっています。



```
MESG    DB "abcdefg",0DH,0AH,'$'
printf("abcdefg¥n");
```

……………アセンブラの場合
復帰改行コード
……………Cの場合
ソースファイル中での文字列の扱いの例
(「abcdefg(復帰改行)」という文字列の場合)
Cプログラム
MS-DOSのテキスト形式
¥n
ライブラリ関数(変換)
0DH, 0AH
¥n
ライブラリ関数(変換)
0DH, 0AH

図6.10　ソースプログラムでの復帰改行コードの扱い例

■ その他の関数

そのほかにも、たいていの処理系には、次に示すような MS-DOS のシステムコールに相当するライブラリ関数がたくさん用意されています。

chdir　…………ディレクトリの変更
mkdir …………ディレクトリの作成
time　…………システムの時間を得る

これらの関数群は、あくまで各処理系独自の機能であって、C 言語自身の機能ではありませんので、その関数が用意されているかいないかは処理系によって異なります。また用意されていても、処理系によっては名前が違う場合もありますので注意が必要です。

C 言語の機能が強力であるのは、このようなライブラリが豊富に用意されているおかげですが、ライブラリを作るのが各メーカーであるために、多少の相違があるのは仕方がないことなのでしょう。

■ ASCIZ 文字列の役割

C 言語と MS-DOS との相性がよく、いろいろな機能を持った関数が簡単に実現できるのは、C 言語における文字列の形式と、MS-DOS のシステムコールで要求される文字列の形(ASCIZ 文字列)が同じであることが大きく関係しています。これは、C のプログラムから MS-DOS へ、MS-DOS から C のプログラムへ、どちらの方向へも文字列をそのままの形で処理できることを意味しています。これは明らかに、MS-DOS の文字列の形式が、C 言語および UNIX との整合性を意識して決定されたためと思われます。

6.3　大きなプログラムの開発

これまでは、ライブラリを中心に C 言語の開発環境について述べてきましたが、ここでは全体的に見た C 言語の開発環境について考えてみましょう。

C 言語で書いたソースファイルは、アセンブラのものに比べてはるかに読みやすいことはいうまでもありません。C 言語では、アセンブラほどレジスタの退避やスタックのレベルに気を遣うことなく、純粋な論理に思考を集中することができます。逆に、マクロアセンブラや高級言語は、このような要求から生まれたといえます。

C 言語には、プログラムやデータを構造化できるという非常に大きなメリットがあります。このことは、構造化がむずかしい形式の高級言語と比べて、プログラムの開発効率を向上させるでしょう。

■ モジュール別プログラミング

C言語は、5章で解説したようなモジュール別プログラミングを行うのに適しています。C言語の関数は、アセンブラのPROC(サブルーチン)にあたりますが、C言語のプログラムはこの関数の集まりとして表され、いくつかの関数により1つのモジュールが構成されます。その際、アセンブラによる開発のところでも述べたように、モジュールの設計をしっかり行って、モジュールの独立性を高めておくことが大切です。そうすることにより、モジュール単体でのデバッグが可能となり、またライブラリを構成することもできるのです。

C言語の処理系の多くは、出力するオブジェクトファイルの形式をMASMの出力するリロケータブルオブジェクトファイル「.OBJ」の形式に合わせているため、Cのプログラムにおいても、モジュール別プログラミングが容易に実現可能です。また、C言語からアセンブラのPROCを呼んだり、ほかのプログラミング言語からC言語の関数を呼んだりするなど、ほかのプログラミング言語とのリンクも比較的容易に行うことができます。

C言語は、コンパイル時にプログラムのチェックをほとんど行いませんが、かなり効率のよいコードを生成します。ですから、実行速度の点から今までアセンブラでなければ不可能だと思われていた処理も、C言語でこなせるものが多いでしょう。よほど高速性を要求するルーチンとか、複雑なI/Oを超高速でアクセスするというような場合を除けば、アセンブラを使う必要はありません。もしどうしても必要な部分があったとしても、速度に大きく影響する、最も内側のループを含む部分に使うだけでよいでしょう。

5章でも述べましたが、モジュール別プログラム開発が可能なことは、大きなプログラムを作成するには絶対的に必要な条件です。大勢の人によってプログラムを開発する場合はとくにそうです。あるまとまった処理ごとのモジュールに分け、そのモジュールごとの開発を行い、モジュールごとに動作を確認しておくのです。それを行わず、大勢の作った動作未確認のそれぞれのモジュールを全部接続してコンパイルし、できあがったプログラムをデバッグするのでは、どこにバグがあるのかを発見することは不可能に近いでしょう。

またモジュール化ができるということは、ほかのモジュールで使われている変数やラベルなどと同じ名前を、自分のモジュールの中だけで使うことができるということです。モジュール化ができないと、ほかのモジュールと同じ名前の変数の内容が書き換えられてしまうことになります。これでは、モジュールの独立性や、ブラックボックス化は困難です。BASICなどのプログラミング言語は、このようなことができないため、モジュール化はほとんど不可能です。もちろん、変数名が重ならないように注意するなどして、ある程度分離することもできるでしょうが、さきにも述べたように、そのようなつまらないことに神経を遣うより、プログラムの論理そのものに思考を集中できる環境でなければよい仕事はできません。

このようなことから、大きなプログラムを開発するには、モジュール別プログラミングによることが必須です。C言語はそのために分割コンパイルが可能であり、モジュール別の開発を最初から想定し

て作られていますので、かなり大きなプログラムの開発にも十分耐えられるでしょう。筆者らの経験からもこれは確信しています。しかし、C 言語のモジュール化の機能も完全ではないため、あくまで各モジュールの設計がしっかりしていなければなりません。前述のプロトタイプ宣言による引数の型のチェック機能を活用するなど、いずれにしても仕様の一貫性を保つことには気を遣う必要があります。

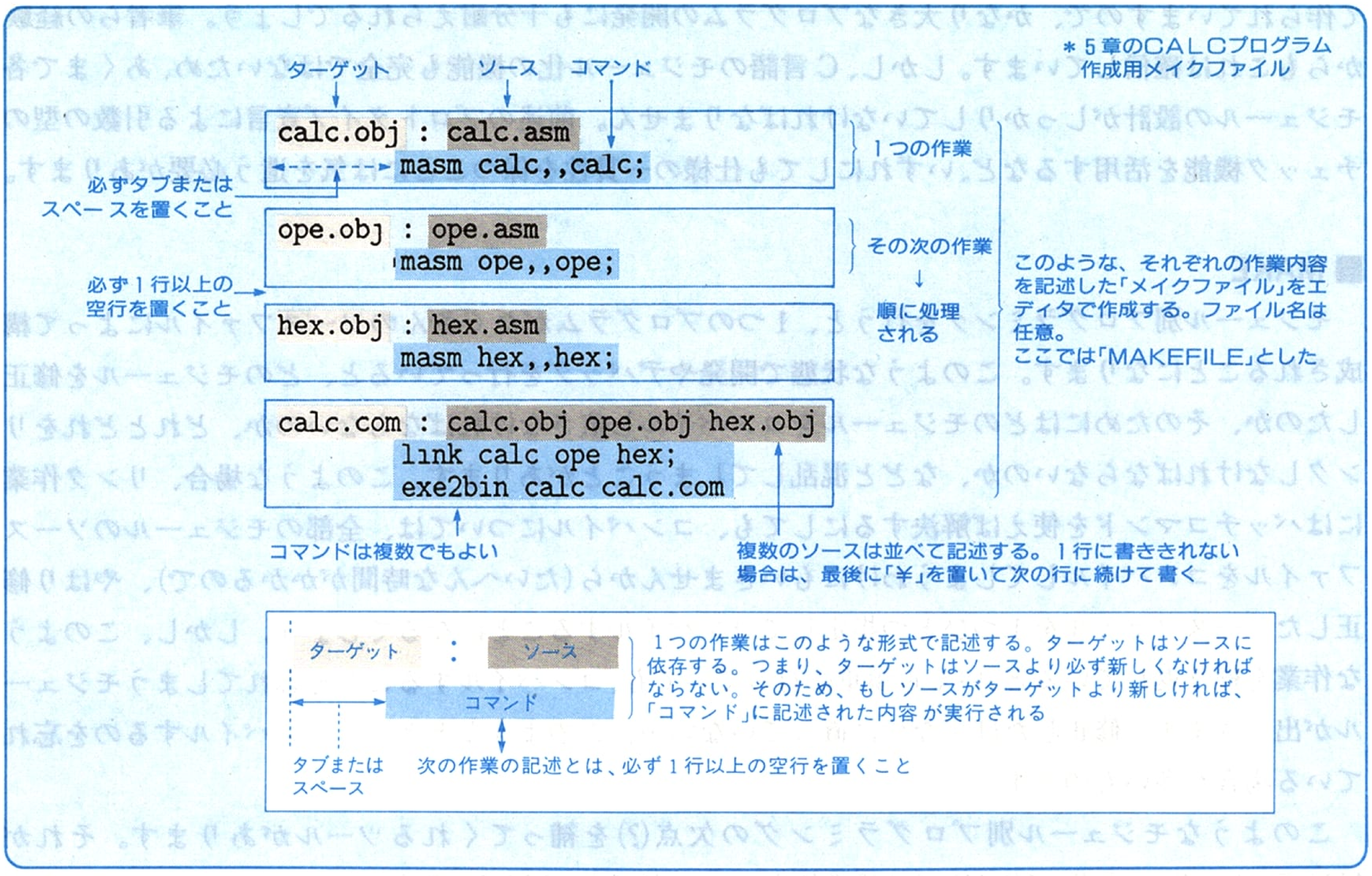
■ MAKE

モジュール別プログラミングを行うと、1つのプログラムがたくさんのソースファイルによって構成されることになります。このような状態で開発やデバッグを行っていると、どのモジュールを修正したのか、そのためにはどのモジュールをコンパイルし直さなければならないのか、どれとどれをリンクしなければならないのか、などと混乱してしまうことがあります。このような場合、リンク作業にはバッチコマンドを使えば解決するにしても、コンパイルについては、全部のモジュールのソースファイルをコンパイルしてしまうわけにもいきませんから(たいへんな時間がかかるので)、やはり修正したソースファイルを1つひとつ指定してコンパイルすることになるでしょう。しかし、このような作業を繰り返していると、いくら慎重に行っていても、コンパイルすることを忘れてしまうモジュールが出てきます。修正したはずなのに直っていない…、このようなときは、コンパイルするのを忘れている場合が多いものです。

このようなモジュール別プログラミングの欠点(?)を補ってくれるツールがあります。それが **MAKE**(メイク)であり、UNIX では標準的なコマンドです。MS-DOS ではバージョン 3.*x* では、その簡易版が付属しています。MAKE は、多くのソースファイルの中で更新(修正)されたファイルがどれであるかを調べ、更新されたものだけを選んで自動的にコンパイルするという機能を持っています。これは一度使ってみると手放せなくなる便利なものです。どのソースファイルをどのように修正しても、そのあとで MAKE コマンドを実行することにより、再コンパイルを必要とする修正されたソースファイルだけがコンパイル(アセンブラの場合はアセンブル)され、そのあとでさらにリンクの作業まで実行してくれるのです。

さて、MAKE はファイルが更新されたことをどうして知るのでしょう。これはファイルの更新日時を比較することによって行われます。オブジェクトファイルの日時がソースファイルの日時より古ければ、最後のコンパイルを行ってからあとにソースファイルが更新されたことがわかります。ファイルの更新された日時を、ファイル情報として持っている MS-DOS だからこそ、このようなことが可能なのです。

MS-DOS バージョン 3.*x* に付属の標準 MAKE を実行するには、実行可能オブジェクトファイルを得るために必要な、各モジュールのソースファイルの構成、それらのソースファイルに対するコンパイルあるいはアセンブルの仕方、オブジェクトファイルのリンクの仕方を書いた**メイクファイル**と呼ばれるファイルを、あらかじめ作成しておかなければなりません(**図 6.11**)。MAKE は、このメイクファイルを参照し、必要な処理を決定し実行します(**図 6.12** 参照)。



```
calc.obj : calc.asm
	masm calc,,calc;

ope.obj : ope.asm
	masm ope,,ope;

hex.obj : hex.asm
	masm hex,,hex;

calc.com : calc.obj ope.obj hex.obj
	link calc ope hex;
	exe2bin calc calc.com
```

ターゲット
ソース
コマンド
＊5章のCALCプログラム作成用メイクファイル
必ずタブまたはスペースを置くこと
1つの作業
その次の作業
順に処理される
必ず1行以上の空行を置くこと
コマンドは複数でもよい
ターゲット
:
ソース
コマンド
タブまたはスペース

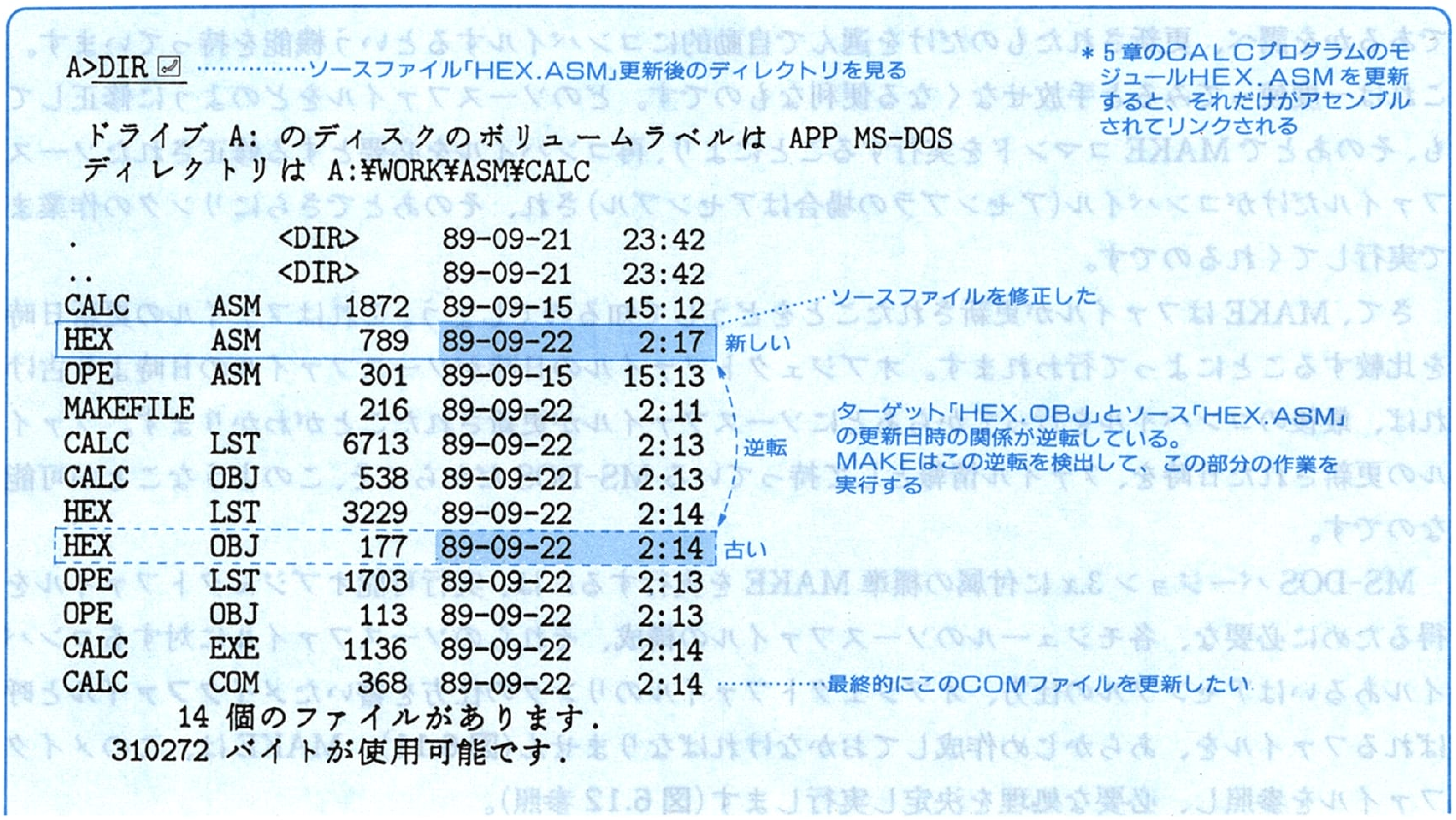
このような、それぞれの作業内容を記述した「メイクファイル」をエディタで作成する。ファイル名は任意。ここでは「MAKEFILE」とした

複数のソースは並べて記述する。1行に書ききれない場合は、最後に「¥」を置いて次の行に続けて書く

1つの作業はこのような形式で記述する。ターゲットはソースに依存する。つまり、ターゲットはソースより必ず新しくなければならない。そのため、もしソースがターゲットより新しければ、「コマンド」に記述された内容が実行される

次の作業の記述とは、必ず1行以上の空行を置くこと

図 6.11　MS-DOS バージョン 3.x に付属の標準 MAKE 用のメイクファイルの例

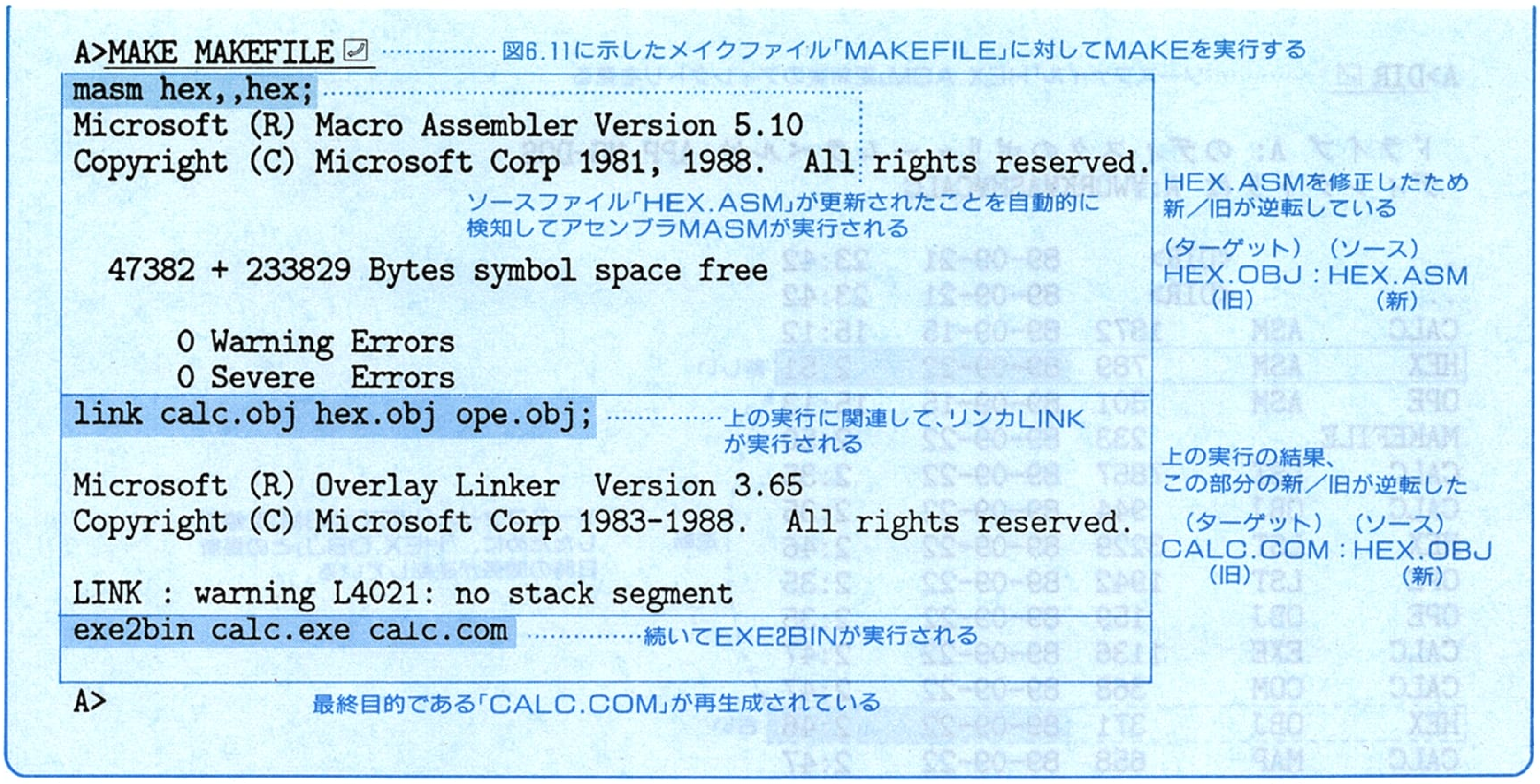


```
A>DIR

 ドライブ A: のディスクのボリュームラベルは APP MS-DOS
 ディレクトリは A:¥WORK¥ASM¥CALC

.            <DIR>     89-09-21   23:42
..           <DIR>     89-09-21   23:42
CALC     ASM      1872  89-09-15   15:12
HEX      ASM       789  89-09-22    2:17
OPE      ASM       301  89-09-15   15:13
MAKEFILE           216  89-09-22    2:11
CALC     LST      6713  89-09-22    2:13
CALC     OBJ       538  89-09-22    2:13
HEX      LST      3229  89-09-22    2:14
HEX      OBJ       177  89-09-22    2:14
OPE      LST      1703  89-09-22    2:13
OPE      OBJ       113  89-09-22    2:13
CALC     EXE      1136  89-09-22    2:14
CALC     COM       368  89-09-22    2:14
        14 個のファイルがあります.
   310272 バイトが使用可能です.
```

ソースファイル「HEX.ASM」更新後のディレクトリを見る
＊5章のCALCプログラムのモジュールHEX.ASMを更新すると、それだけがアセンブルされてリンクされる
ソースファイルを修正した
新しい
逆転
古い
最終的にこのCOMファイルを更新したい

ターゲット「HEX.OBJ」とソース「HEX.ASM」の更新日時の関係が逆転している。MAKEはこの逆転を検出して、この部分の作業を実行する

―図 6.12―（次ページに続く）



```
A>MAKE MAKEFILE ⏎
masm hex,,hex;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.

  47382 + 233829 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
link calc.obj hex.obj ope.obj;

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

LINK : warning L4021: no stack segment
exe2bin calc.exe calc.com

A>
```

図6.11に示したメイクファイル「MAKEFILE」に対してMAKEを実行する

ソースファイル「HEX.ASM」が更新されたことを自動的に検知してアセンブラMASMが実行される

HEX.ASMを修正したため新／旧が逆転している

（ターゲット） （ソース）
HEX.OBJ : HEX.ASM
（旧） （新）

上の実行に関連して、リンカLINKが実行される

上の実行の結果、この部分の新／旧が逆転した

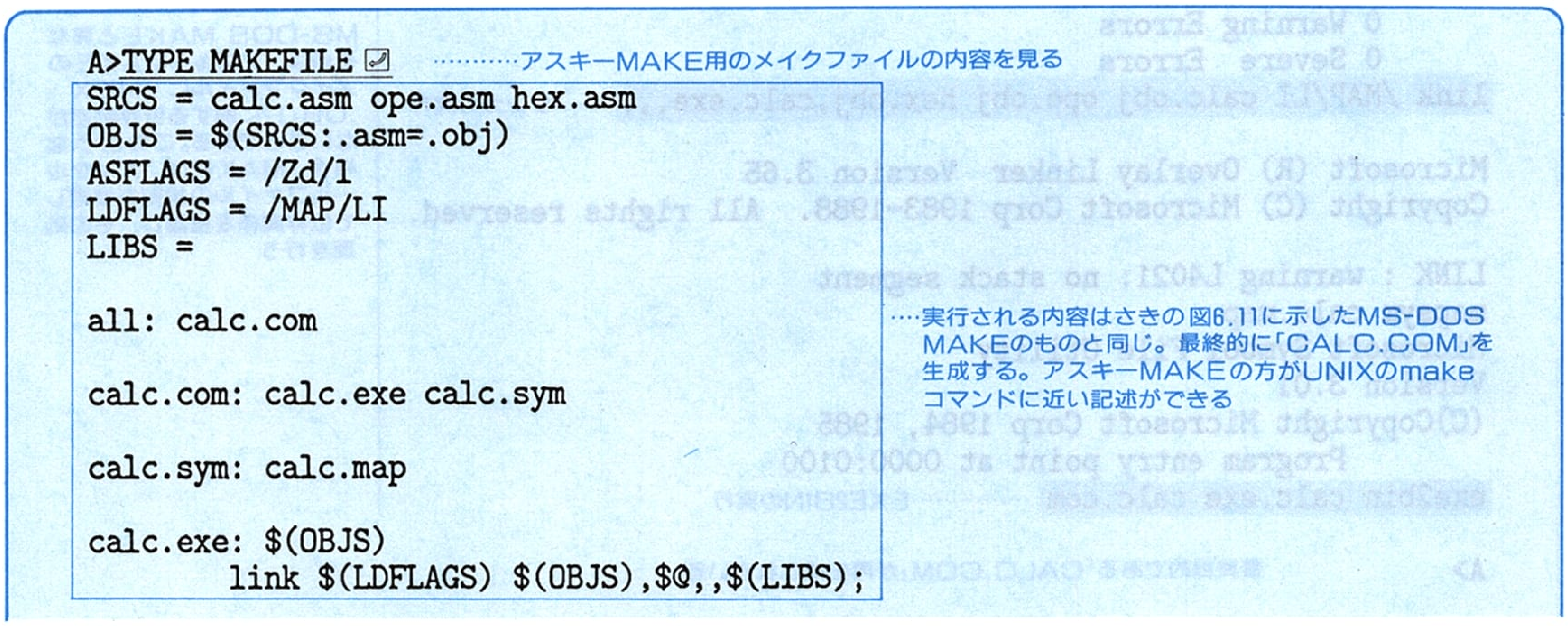
（ターゲット） （ソース）
CALC.COM : HEX.OBJ
（旧） （新）

続いてEXE2BINが実行される

最終目的である「CALC.COM」が再生成されている

図 6.12 MS-DOS の標準 MAKE の実行例

MAKE は、MS-DOS のバージョン 3.*x* では標準で用意されていますが、機能的には不満が残ります（それでもバッチファイルで行うよりははるかに有効だが）。MS-DOS 上の MAKE には、そのほかに市販されている製品もありますので、**図 6.13** にその例（ここでは『MS-DOS SOFTWARE TOOLS 3』* におさめられている make プログラムを取り上げる）を示します。これは図 6.12 と同じことを実行しています。



```
A>TYPE MAKEFILE ⏎
SRCS = calc.asm ope.asm hex.asm
OBJS = $(SRCS:.asm=.obj)
ASFLAGS = /Zd/l
LDFLAGS = /MAP/LI
LIBS =

all: calc.com

calc.com: calc.exe calc.sym

calc.sym: calc.map

calc.exe: $(OBJS)
        link $(LDFLAGS) $(OBJS),$@,,$(LIBS);
```

アスキーMAKE用のメイクファイルの内容を見る

実行される内容はさきの図6.11に示したMS-DOS MAKEのものと同じ。最終的に「CALC.COM」を生成する。アスキーMAKEの方がUNIXのmakeコマンドに近い記述ができる

— 図 6.13 —（次ページに続く）

---

* 『MS-DOS SOFTWARE TOOLS 3』 アスキー出版局刊 1988。make プログラム自体は単体製品ではない。

```
A>DIR⏎

 ドライブ A: のディスクのボリュームラベルは APP MS-DOS
 ディレクトリは A:¥WORK¥ASM¥CALC

.            <DIR>      89-09-21    23:42
..           <DIR>      89-09-21    23:42
CALC     ASM      1872  89-09-15    15:12
HEX      ASM       789  89-09-22     2:51
OPE      ASM       301  89-09-15    15:13
MAKEFILE           233  89-09-22     2:50
CALC     LST      7857  89-09-22     2:35
CALC     OBJ       944  89-09-22     2:35
HEX      LST      3229  89-09-22     2:46
OPE      LST      1942  89-09-22     2:35
OPE      OBJ       159  89-09-22     2:35
CALC     EXE      1136  89-09-22     2:47
CALC     COM       368  89-09-22     2:47
HEX      OBJ       371  89-09-22     2:46
CALC     MAP       658  89-09-22     2:47
CALC     SYM       180  89-09-22     2:47
        16 個のファイルがあります.
   304128 バイトが使用可能です.

A>MAKE⏎
masm /Zd/l hex.asm,hex.obj;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


   47356 + 170591 Bytes symbol space free

       0 Warning Errors
       0 Severe  Errors
link /MAP/LI calc.obj ope.obj hex.obj,calc.exe,,;

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

LINK : warning L4021: no stack segment
mapsym calc.map
Microsoft Symbol File Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
        Program entry point at 0000:0100
exe2bin calc.exe calc.com

A>
```

A>DIR ……………ソースファイル「HEX.ASM」更新後のディレクトリを見る

HEX ASM 789 89-09-22 2:51 新しい

HEX OBJ 371 89-09-22 2:46 古い

逆転

ソースファイル「HEX.ASM」を修正したために、「HEX.OBJ」との更新日時の関係が逆転している

A>MAKE …………アスキーMAKEを実行する。メイクファイルの指定をしない場合は、「MAKEFILE」が自動的に指定される

masm /Zd/l hex.asm,hex.obj; ……………「HEX.ASM」をアセンブルする

link /MAP/LI calc.obj ope.obj hex.obj,calc.exe,,; ……リンカの実行

exe2bin calc.exe calc.com ……………EXE2BINの実行

MS-DOS MAKEと異なる点は、アスキーMAKEのメイクファイルに「HEX.OBJ」に関する依存記述がないことである。このような場合、MAKEは、メイクルールファイルの規則を参照して依存関係を推論し、その処理を行う

最終目的である「CALC.COM」が再生成されている

図 6.13　市販 MAKE（アスキーMAKE）による実行例

6.4　C 言語によるプログラム作成実習

これまでの解説で、C 言語によるソフトウェア開発環境がどのようなものか、その雰囲気はつかめたのではないかと思います。そこで本節では、C 言語によるプログラムを作成し、実際の C プログラミングの感触を体験してみましょう。作成するプログラムは、1 章でアセンブラにより作成した、ファイル属性変更プログラム CHMOD に、システム属性のサポートを追加したプログラム「chmod」です。このプログラムの作成において、本章で解説してきたことの一部を実際に見ることができるでしょう。

ここでの実行例は Microsoft　C バージョン 5.1 を標準として使っています。ほかの処理系を使う場合や、各バージョンによっては相違があると思われますので、各自でそれらの部分をよく確認してください。なお、7 章でも C 言語を使ってプログラムを作成しています。参考にしてください。

■ ファイル属性変更プログラムの作成

1 章で作成した CHMOD プログラムを C 言語で作成してみます。このプログラムは、1 章のプログラムとまったく同じ動作をするものですが、1 章ではサポートしていなかった、システムファイルの属性についてもサポートしています。プログラムの操作は 1 章のものと同じですので、そちらを参照してください。また、追加されたシステム属性の操作は次のように行います。

A＞CHMOD　ファイル名/S ⏎………システムファイルにする
A＞CHMOD　ファイル名/U ⏎………システムファイルを普通のファイルにする

chmod 全体は 3 つのファイル(1 つのヘッダファイルを共有する 2 つのモジュール)から構成されています。

chmod.c ……… メインモジュール
attrib.c ……… システムコールを直接使って、ファイル属性を操作するモジュール
attrib.h ……… 文字列に数値を定義するヘッダファイル

さっそくこれらのソースファイルを示しましょう(**リスト 6.1、リスト 6.2、リスト 6.3**)。

メインモジュール

```
/*
 * chmod.c       - change file attributes command
 */
#include <stdio.h>
#include <string.h>

#include "attrib.h"

USHORT err = 0;
USHORT doserrno = 0;

void dosperror(const char *name);
void prperm(const char *name, const ATTRIB attr);

static char _usage[] =
        "Usage : chmod <file>{/<switch>}...¥n"
        " ¥"chmod¥" changes specfed file's attributes.¥n"
        "/<switch> is ¥n"
        "  /R   set to read only¥n"
        "  /W   set to read/write¥n"
        "  /H   set to hidden file¥n"
        "  /N   set to visible file¥n"
        "  /S   set to system file¥n"
        "  /U   set to user file¥n"
        "  /?   force print attributes¥n"
        " when no switches specified,¥n"
        " print attributes of the file¥n"
        ;

#define usage() fputs(_usage, stderr)

void dosperror(const char *name)
{
        switch(doserrno) {
        case 0x0000:
                fputs("No error exist.¥n", stderr);
                break;
        case 0x0001:
                fprintf( stderr,
                        "INTERNAL ERROR %s: (1) Invalid function.¥n"
                        "Please tell the author of " __FILE__ "¥n",
                        name );
                abort();
        case 0x0002:
                fprintf(stderr, "ERROR %s: (2) No such file.¥n", name);
                break;
        case 0x0003:
                fprintf(stderr, "ERROR %s: (3) No such path.¥n", name);
                break;
        case 0x0005:
                fprintf(stderr, "ERROR %s: (5) Access denied.¥n", name);
                break;
```

このプログラムの使い方の表示

エラーメッセージ表示関数

プログラムが正しければ、けっして実行されないはず。その際のエラーメッセージ

コンパイラが、ソースファイルの名前に置き換えてくれる

標準エラー出力へ出力する

— リスト 6.1 — (次ページ以下に続く)

```
        default:
                fprintf( stderr,
                        "INTERNAL ERROR %s: (%u) Can't recognized error.¥n"
                        "Please tell the author of " __FILE__ "¥n",
                        name, doserrno );
                abort();
        }
}
```

原因不明のエラーが発生した場合のメッセージ。通常はけっして実行されない

```
void prperm(const char *name, const ATTRIB attr)
{
#ifdef DEBUG
        printf("%s: %2X¥n", name, attr);
#endif
        printf( "%s : %s%s%s%s¥n", name,
                (attr & A_RDONLY) ? "Read only " : "Read/Write ",
                (attr & A_HIDDEN) ? "Hidden ": "",
                (attr & A_SYSTEM) ? "System ": "",
                (attr & A_DIRECT) ? "Directory" :
                ((attr & A_VOLUME) ? "Volume" : "File") );
}
```

たとえば、Microsoft Cでは
A>CL -DDEBUG chmod.c

デバッグ時に使用する。得られた属性値をそのまま表示するためのもの。コンパイル時にオプションを指定するとDEBUGが定義され、この部分が有効になる。指定しない場合は、この部分は無効になり、コンパイルされない

ファイル属性値による意味をファイル名とともに表示する関数

```
void main(int argc, char *argv[])
{
        char *name;
        register char *p;
        int disp, cerr;
        ATTRIB mask,attr;

        if (argc < 2) {
                usage();
                exit(1);
        }

        while (--argc > 0) {
                cerr = 0;
                p = strchr(*++argv, '/');
                if (p == NULL) {
                        if (getattr(*argv, &attr) != 0) {
                                dosperror(*argv);
                                err |= 2;
                        }
                        prperm(*argv, attr);
                        continue;
                }
                name = *argv;
                *p = '¥0';
                mask = attr = A_NONE;
                disp = 0;
```

main関数。Cプログラムの実行はここから始まる

argc=2

コマンドラインは少なくとも chmod ファイル名/x の2つがなければならない。そうでない場合は「使い方」の表示へ

argv[0]　argv[1]

ただし、MS-DOSバージョン2.xではセットされない

ファイル名の終わりを捜す

ファイル属性を得る。見つからなければ、エラーコードを返す

スイッチを指定しない場合は、そのファイルの現在の属性を表示する

```
        for (;;) {
                switch (*++p) {
                case ’R’:
                case ’r’:
                        mask |= A_RDONLY;
                        attr |= A_RDONLY;
                        p++;
                        break;
                case ’W’:
                case ’w’:
                        mask |= A_RDONLY;
                        attr &= ~A_RDONLY;
                        p++;
                        break;
                case ’H’:
                case ’h’:
                        mask |= A_HIDDEN;
                        attr |= A_HIDDEN;
                        p++;
                        break;
                case ’N’:
                case ’n’:
                        mask |= A_HIDDEN;
                        attr &= ~A_HIDDEN;
                        p++;
                        break;
                case ’S’:
                case ’s’:
                        mask |= A_SYSTEM;
                        attr |= A_SYSTEM;
                        p++;
                        break;
                case ’U’:
                case ’u’:
                        mask |= A_SYSTEM;
                        attr &= ~A_SYSTEM;
                        p++;
                        break;
                case ’?’:
                        disp = 1;
                        p++;
                        break;
                case ’/’:
                case ’¥0’:
                        fprintf( stderr,
                                "%s: No switch¥n", name );
                        cerr = 1;
                        err |= 1;
                        break;
```

スイッチに従って変更すべき属性の場所と値を設定する

変更後に属性を表示するように設定する

スイッチ文字がなかったエラーを表示して解析を続ける

```
                        default:
                                fprintf( stderr,
                                          "%s/%c: Bad switch¥n", name, *p );
                                cerr = 1;
                                err |= 1;
                                p++;
                                break;
                        }
                        if (*p == '¥0')
                                break;
                        if (*p != '/') {
                                fprintf(stderr, "%s: '/' is expected¥n", name);
                                cerr = 1;
                                err |= 1;
                                p--;
                        }
                }
                if (!cerr) {
                        if (attrib(*argv, mask, &attr) != 0) {
                                dosperror(*argv);
                                err |= 2;
                        } else if (disp)
                                prperm(*argv, attr);
                }
        }
        exit(err);
}
/* end of chmod.c */
```

スイッチ文字が正しくなかった。エラーを表示して解析を続ける

属性値を変更する

リスト 6.1　メインモジュール chmod.c のソースプログラム

システムコール・モジュール

```
/*
 * attrib.c      - change MS-DOS file attributes function
 */

#include <stdio.h>
#include "attrib.h"
#if defined(MSC) || defined(__TURBOC__)
# include <dos.h>
#else
# error(Please insert your compiler's include file like <dos.h> here.)
#endif

extern USHORT doserrno;
```

［このモジュールは、ファイル属性を取得、設定する関数である。CからMS-DOSのシステムコールを直接利用しているが、この方法は処理系にかなり依存する］

処理系に依存する部分をマクロとプリプロセッサ制御文で避ける

TURBO Cがデフォルトで定義するマクロシンボル

エラーメッセージを出力してコンパイルを中止させる制御文

— リスト 6.2 — (次ページに続く)

```
/*
 * change file attributes
 */
USHORT attrib(const char *path, const ATTRIB mask, PATTRIB attr)
{
        union REGS reg;

        reg.x.ax = 0x4300;
        reg.x.dx = (USHORT)path;
        intdos(&reg, &reg);
        if (reg.x.cflag & 1) {
                doserrno = reg.x.ax;
                return doserrno;
        }
        *attr = ((reg.x.cx & ~mask) | (mask & *attr));
        if (*attr != reg.x.cx) {
                reg.x.ax = 0x4301;
                reg.x.cx = *attr;
                intdos(&reg, &reg);
                if (reg.x.cflag & 1) {
                        doserrno = reg.x.ax;
                        return doserrno;
                }
        }
        return 0;
}
/* end of attrib.c */
```

CPUのレジスタにシステムコールの値をセットする構造体

ファンクションナンバー

「取得」のパラメータ

フラグレジスタの最下位ビットはキャリーフラグ

指定したファイルの属性を得るシステムコール
解説は図6.6参照

キャリーフラグが「1」でなければ、CXレジスタにファイル属性が返されている

「設定」のパラメータ

ファンクションナンバー

指定したファイルに属性を設定するシステムコール

注）このプログラムはスモールモデルでしか動作しません。ラージモデルではDSレジスタも設定する必要があります。

リスト6.2　システムコールモジュールattrib.cのソースプログラム

```
/*
 * attrib.h     - MS-DOS file attributes
 */

#define A_NONE (ATTRIB)0x0000

#define A_RDONLY (ATTRIB)0x0001
#define A_HIDDEN (ATTRIB)0x0002
#define A_SYSTEM (ATTRIB)0x0004
#define A_VOLUME (ATTRIB)0x0008
#define A_DIRECT (ATTRIB)0x0010
#define A_ARCHIV (ATTRIB)0x0020

typedef unsigned short USHORT;
typedef USHORT ATTRIB;
typedef ATTRIB *PATTRIB;

USHORT attrib(const char *filename, const ATTRIB mask, PATTRIB attr);
#define getattr(name, attr) ¥
        attrib(name, A_NONE, attr)
```

ヘッダファイル

ファイル属性値を定義しておくヘッダ。このようにいろいろな値や式を定義しておくとプログラムがわかりやすくなり、間違いを少なくするために役立つ

attrib関数のプロトタイプ宣言。この宣言により、attrib関数の呼び出しの引数の数や型がチェックされる

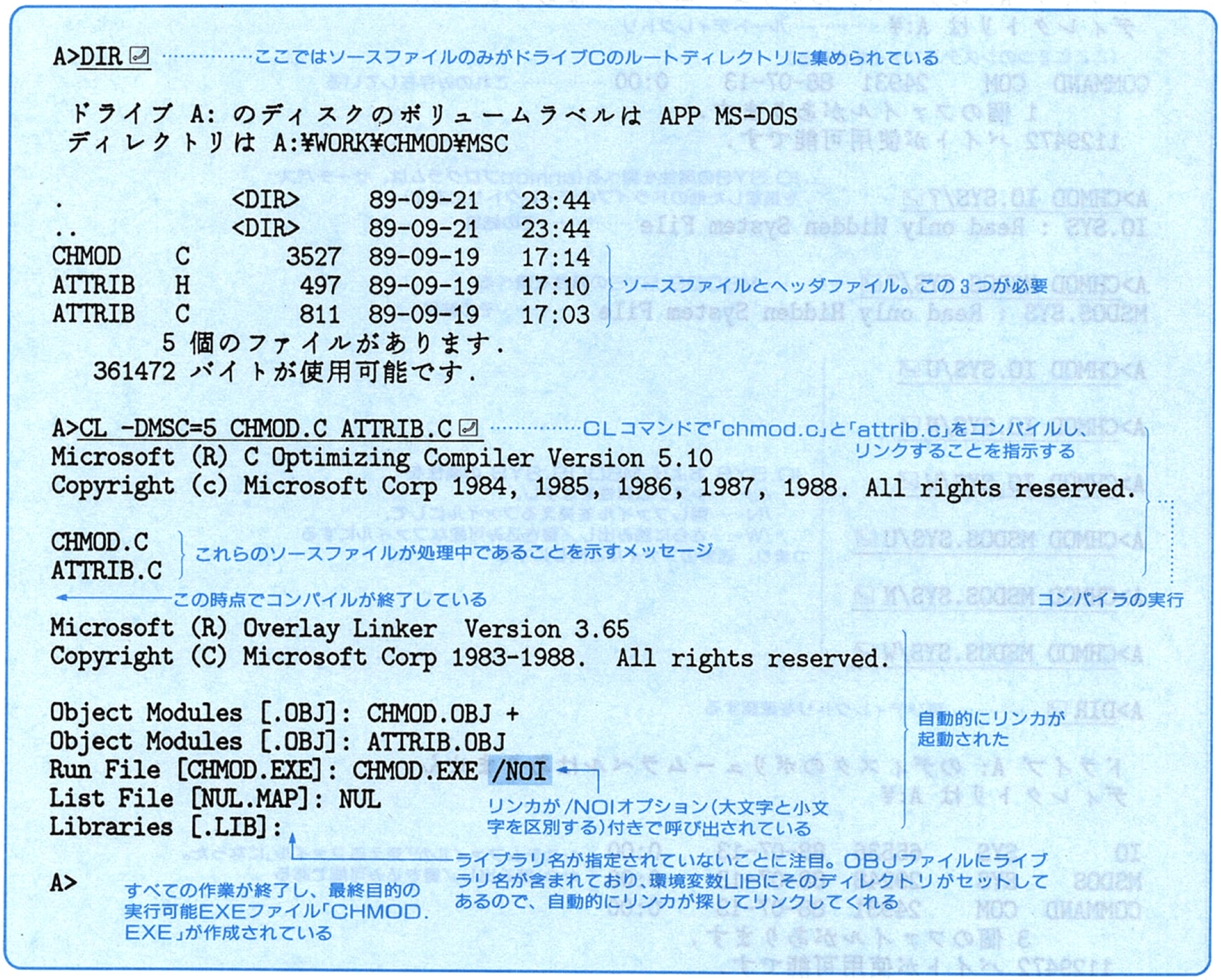
リスト6.3　ファイル属性値を定義しておくヘッダファイルattrib.h

まずエディタを使って、それぞれのファイル名でこれら3つのソースファイルを作成します。こんな簡単なプログラムをなぜわざわざ3つに分けるのかについては、本章や5章で繰り返し述べているとおりです。

これら3つのソースファイルができあがると、次はコンパイルです。**図6.14**にその手順を示します。MS-Cでは、この作業にCL(Compile & Link)というコマンドが用意されていて、これを使うと、コンパイルを実行したあと、リンクまで一度に実行してくれます。ただしこの場合には、環境変数として、

SET INCLUDE＝ヘッダファイルのあるディレクトリ名
SET LIB＝ライブラリファイルのあるディレクトリ名

の2つを登録しておく必要があります(AUTOEXEC.BATに登録しておけばよい)。



```
A>DIR↵

 ドライブ A: のディスクのボリュームラベルは APP MS-DOS
 ディレクトリは A:¥WORK¥CHMOD¥MSC

.            <DIR>      89-09-21   23:44
..           <DIR>      89-09-21   23:44
CHMOD    C        3527  89-09-19   17:14
ATTRIB   H         497  89-09-19   17:10
ATTRIB   C         811  89-09-19   17:03
        5 個のファイルがあります.
   361472 バイトが使用可能です.

A>CL -DMSC=5 CHMOD.C ATTRIB.C↵
Microsoft (R) C Optimizing Compiler Version 5.10
Copyright (c) Microsoft Corp 1984, 1985, 1986, 1987, 1988. All rights reserved.

CHMOD.C
ATTRIB.C

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Object Modules [.OBJ]: CHMOD.OBJ +
Object Modules [.OBJ]: ATTRIB.OBJ
Run File [CHMOD.EXE]: CHMOD.EXE /NOI
List File [NUL.MAP]: NUL
Libraries [.LIB]:

A>
```

ここではソースファイルのみがドライブCのルートディレクトリに集められている

ソースファイルとヘッダファイル。この3つが必要

CLコマンドで「chmod.c」と「attrib.c」をコンパイルし、リンクすることを指示する

コンパイラの実行

これらのソースファイルが処理中であることを示すメッセージ

この時点でコンパイルが終了している

自動的にリンカが起動された

リンカが/NOIオプション(大文字と小文字を区別する)付きで呼び出されている

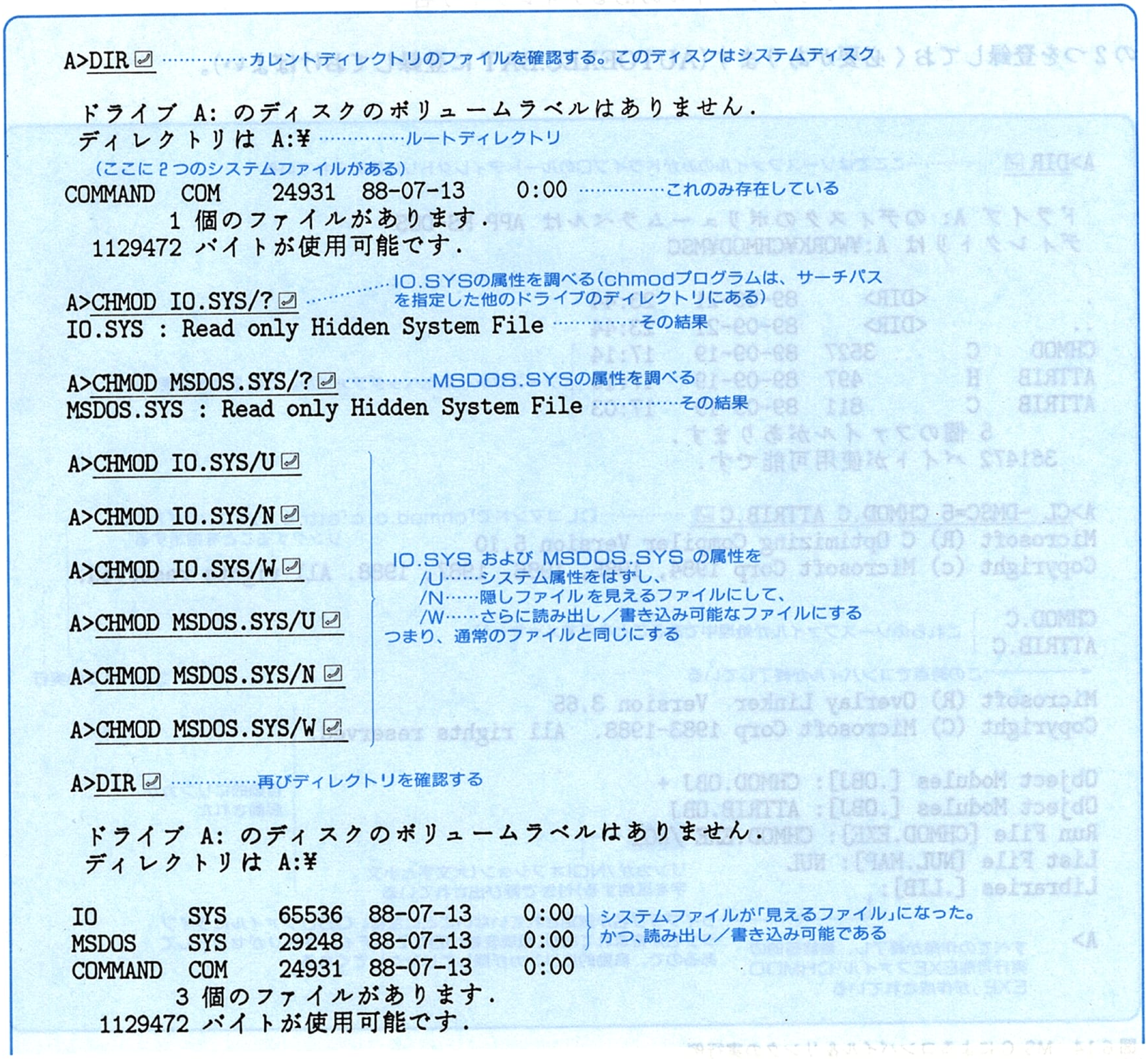
ライブラリ名が指定されていないことに注目。OBJファイルにライブラリ名が含まれており、環境変数LIBにそのディレクトリがセットしてあるので、自動的にリンカが探してリンクしてくれる

すべての作業が終了し、最終目的の実行可能EXEファイル「CHMOD.EXE」が作成されている

図6.14　MS-Cによるコンパイル＆リンクの実行例

以上の作業で、chmodの実行可能なオブジェクトファイル「CHMOD.EXE」(ファイル名は大文字になる)が完成しました。もし、コンパイルやリンクの途中でエラーメッセージが表示され、実行が止まってしまった場合は、ソースファイルやコンパイルのコマンドラインなどに、何らかのミスがあると思われます。誤りを見つけ出して修正したあと、もう一度コンパイルおよびリンクを実行してみてください。

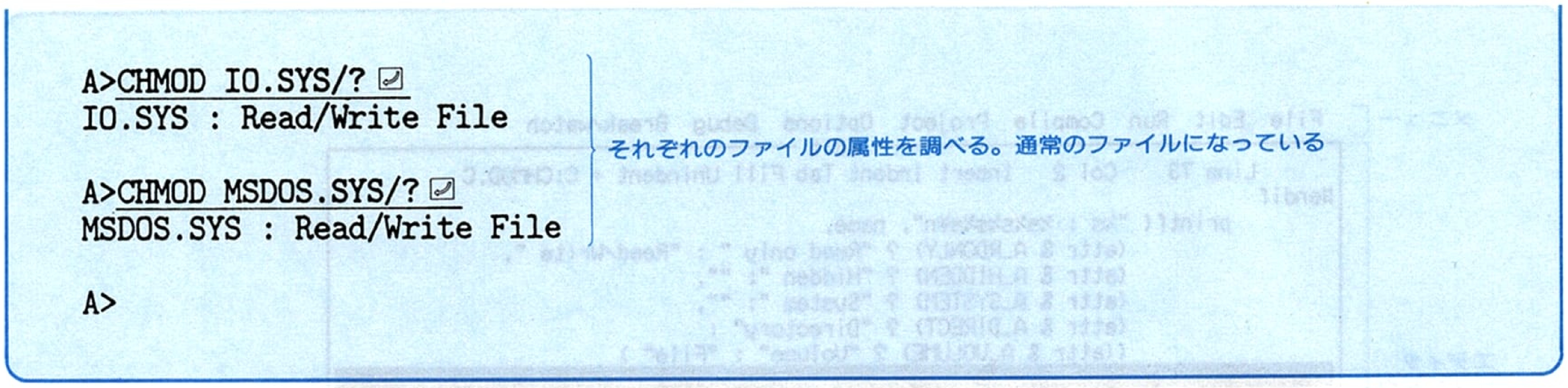
では、完成したchmodプログラムをさっそく使ってみましょう(図6.15)。システムファイルの属性がセットされているIO.SYSおよびMSDOS.SYSを、読み出し／書き込み可能で、隠されていない普通のファイルに変更してみます。



```
A>DIR↵

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

COMMAND  COM     24931  88-07-13     0:00
          1 個のファイルがあります.
  1129472 バイトが使用可能です.

A>CHMOD IO.SYS/?↵
IO.SYS : Read only Hidden System File

A>CHMOD MSDOS.SYS/?↵
MSDOS.SYS : Read only Hidden System File

A>CHMOD IO.SYS/U↵

A>CHMOD IO.SYS/N↵

A>CHMOD IO.SYS/W↵

A>CHMOD MSDOS.SYS/U↵

A>CHMOD MSDOS.SYS/N↵

A>CHMOD MSDOS.SYS/W↵

A>DIR↵

 ドライブ A: のディスクのボリュームラベルはありません.
 ディレクトリは A:¥

IO       SYS     65536  88-07-13     0:00
MSDOS    SYS     29248  88-07-13     0:00
COMMAND  COM     24931  88-07-13     0:00
          3 個のファイルがあります.
  1129472 バイトが使用可能です.
```

……カレントディレクトリのファイルを確認する。このディスクはシステムディスク
……ルートディレクトリ
(ここに2つのシステムファイルがある)
……これのみ存在している
……IO.SYSの属性を調べる(chmodプログラムは、サーチパスを指定した他のドライブのディレクトリにある)
……その結果
……MSDOS.SYSの属性を調べる
……その結果
IO.SYS および MSDOS.SYS の属性を
/U……システム属性をはずし、
/N……隠しファイルを見えるファイルにして、
/W……さらに読み出し／書き込み可能なファイルにする
つまり、通常のファイルと同じにする
……再びディレクトリを確認する
システムファイルが「見えるファイル」になった。
かつ、読み出し／書き込み可能である

— 図6.15 —(次ページに続く)



```
A>CHMOD IO.SYS/?⏎
IO.SYS : Read/Write File

A>CHMOD MSDOS.SYS/?⏎
MSDOS.SYS : Read/Write File

A>
```

それぞれのファイルの属性を調べる。通常のファイルになっている

図 6.15　完成した chmod プログラムの実行例

6.5　より進んだプログラミング環境

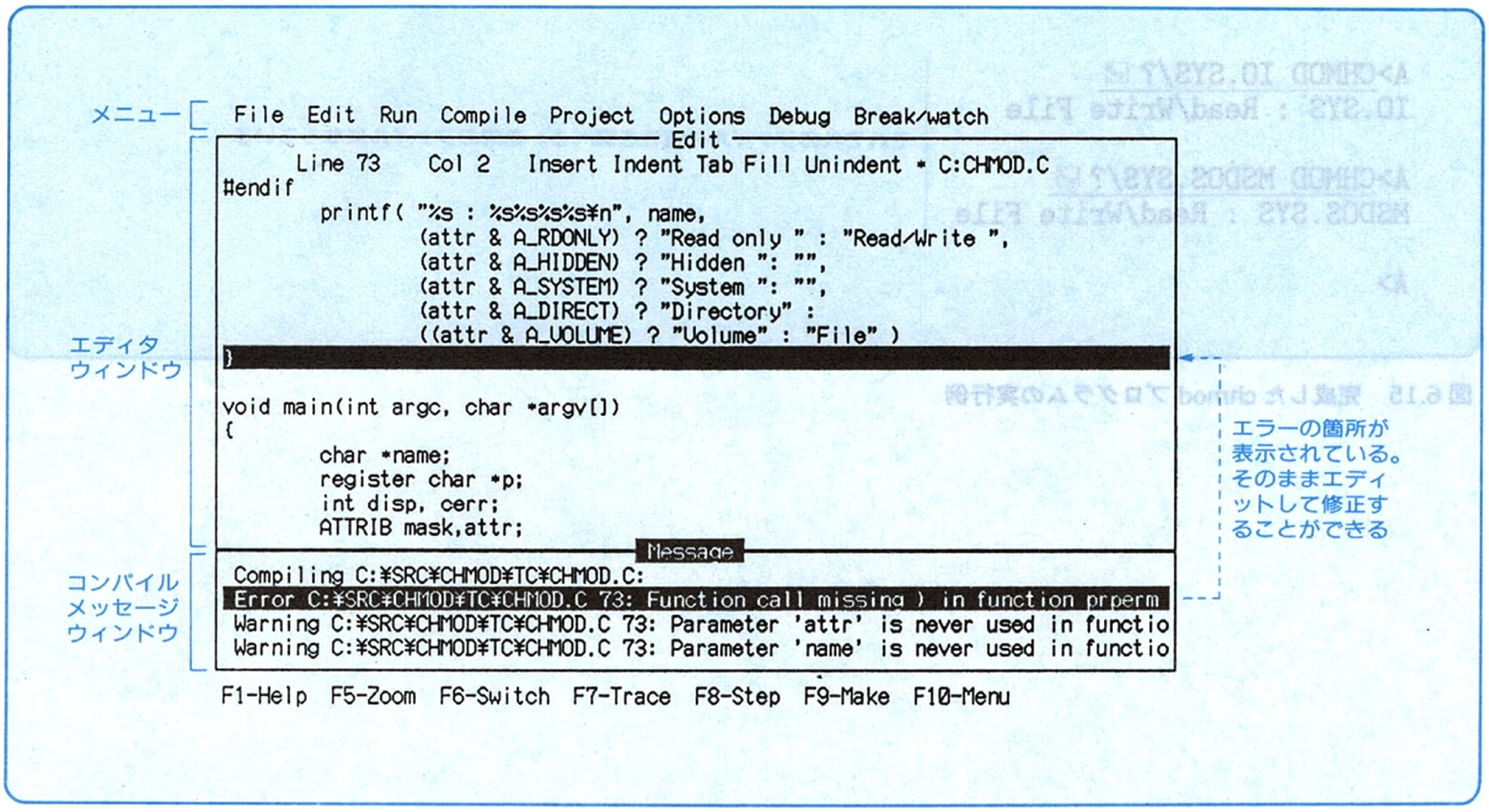
最近の処理系には、統合プログラミング環境を実現したものや、5 章でも紹介した CODEVIEW を備えたものが増えてきました。本章の最後に、これらのより進んだプログラミング環境を紹介しましょう。

■ 統合化されたプログラミング環境

エディタやコンパイラ、デバッガといった個々のプログラミングツールが新世代のツールに移行する一方、それらをひとつのまとまった環境として提供する統合型ツールへの動きが進行しています。Quick C や Turbo C はその代表例であり、広く使われています。

統合化環境では個々のツールの機能は専用ツールに比べて比較的単純ですが、各ツールの連携プレーによりプログラム作成を強力にサポートします。たとえば、内蔵のエディタでプログラムを作成し、エディタのメニューからコンパイルを指示することができます。そこで文法エラーが検出されると、エディタに戻って自動的にエラーの箇所にカーソルが移動するといった具合いです。コンパイルは高速なのでプログラムを作成、修正後すぐに実行させて結果を確認することができます。さらに CODEVIEW のような優れたユーザーインターフェイスを持ったデバッガを内蔵しており、ブレークポイントの設定や変数の値の確認なども可能となっています。

このように各ツールが有機的に結び付けられているので、プログラム作成のサイクルを大幅に短縮できます(図 6.16 参照)。



```
 File  Edit  Run  Compile  Project  Options  Debug  Break/watch
─────────────────────────── Edit ──────────────────────────────
      Line 73    Col 2   Insert Indent Tab Fill Unindent * C:CHMOD.C
#endif
        printf( "%s : %s%s%s%s¥n", name,
                (attr & A_RDONLY) ? "Read only " : "Read/Write ",
                (attr & A_HIDDEN) ? "Hidden ": "",
                (attr & A_SYSTEM) ? "System ": "",
                (attr & A_DIRECT) ? "Directory" :
                ((attr & A_VOLUME) ? "Volume" : "File" )
}

void main(int argc, char *argv[])
{
        char *name;
        register char *p;
        int disp, cerr;
        ATTRIB mask,attr;
────────────────────────── Message ─────────────────────────────
 Compiling C:¥SRC¥CHMOD¥TC¥CHMOD.C:
 Error C:¥SRC¥CHMOD¥TC¥CHMOD.C 73: Function call missing ) in function prperm
 Warning C:¥SRC¥CHMOD¥TC¥CHMOD.C 73: Parameter 'attr' is never used in functio
 Warning C:¥SRC¥CHMOD¥TC¥CHMOD.C 73: Parameter 'name' is never used in functio

F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

エラーの箇所が表示されている。そのままエディットして修正することができる

図6.16　Turbo　Cの統合プログラミング環境

統合化プログラミング環境は、プログラム開発の効率を向上させることが目的ですがそれだけではありません。プログラムを修正してから実行するまでの手順がスムーズであることから実験的なプログラミングに効果的であり、また、デバッガでプログラムの動作を逐一見守ることができるため、プログラミング言語の習得にも力を発揮します。プログラム作成技術の向上にも大きな役割を果たすことでしょう。

■ ソースコードデバッガ　CODEVIEW

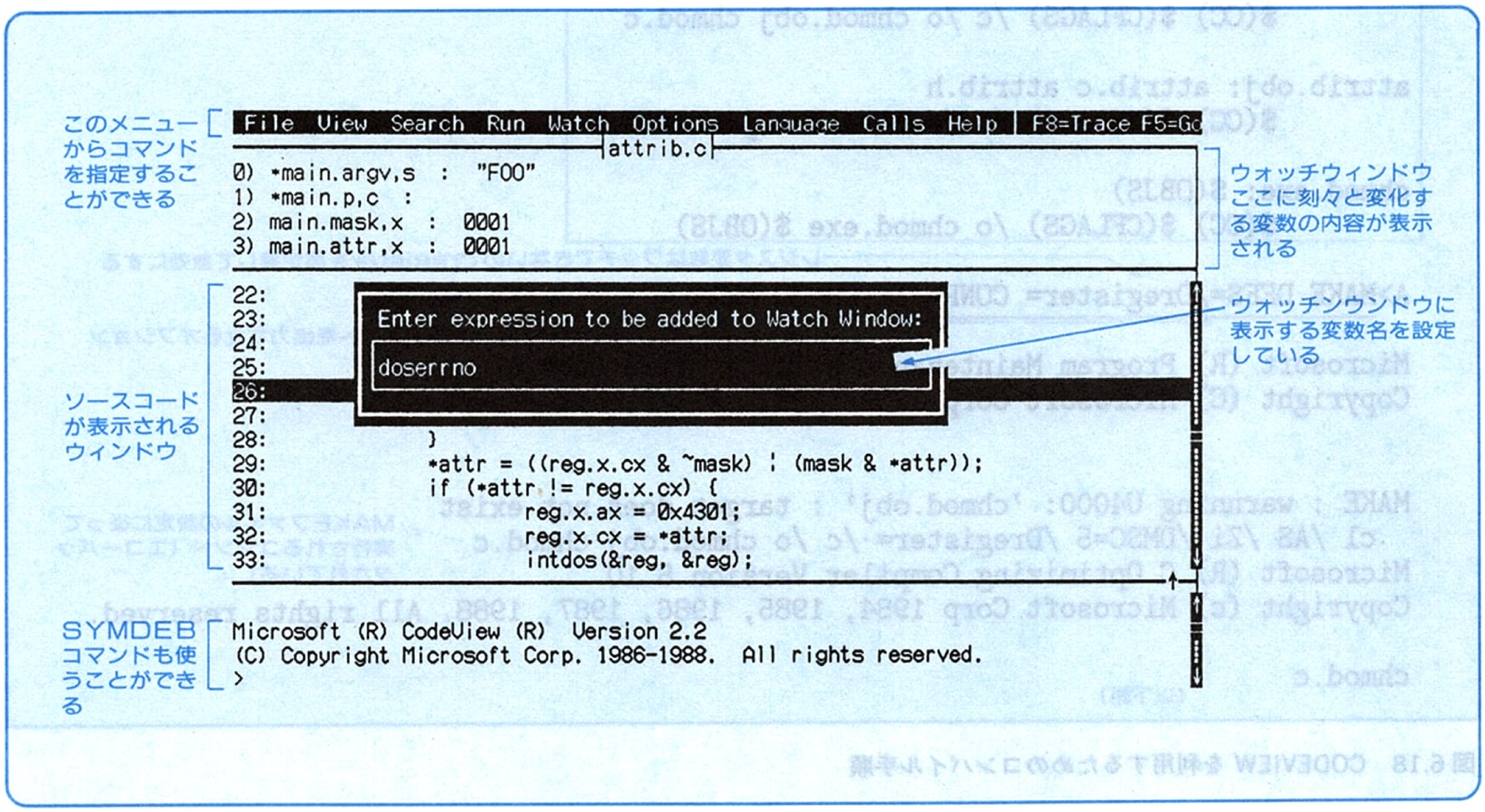
CODEVIEWやTurbo-DebuggerはCなど高級言語のデバッグを対話的に行うことを可能にしたデバッガです。DEBUGまでは基本的には8ビット時代のデバッガと同程度の機能でしたが、これらは16ビットの本領を発揮した新世代のデバッガといえるでしょう。

旧世代のデバッガではマシン語の知識は不可欠であり、高級言語のプログラムのデバッグには図6.5のようにプログラム中にデバッグのためのコードを埋め込むくらいしか有効な手段はありませんでした。マシン語の知識があればSYMDEBなどのデバッガを使うことも可能ですが、コンパイルされたあとのマシン語コードを対象としなければならず、多少の熟練を要します。

CODEVIEWを使えば、たとえば図6.17のようにコメントなどを含めたC言語のソースコードを対象にデバッグすることができます。まるでスクリーンエディタを操作するような感覚でブレークポイントを設定したり、ステップ実行したりすることが可能になるのです。

このスクリーンエディタのような操作感覚こそ、CODEVIEW を新世代のデバッガと呼ぶ理由でもあります。メニューからコマンドを選択して実行でき、ファンクションキーでブレークポイントやステップ実行を指示できるなど、ユーザーインターフェイスが格段に向上しています*。

また、デバッガとしての機能も高級言語を扱えるように拡張されています。たとえば、ローカル変数の内容を変数名を使って確認することができ（SYMDEB ではグローバル変数のみ可能）、さらに、プログラムの実行中に刻々と変化する変数の値をウォッチウィンドウに表示するウォッチ機能や、変数の値が指定した条件にあてはまるように変更されたらブレークするウォッチポイント機能など、プログラムのデバッグに有効な数多くの機能が加えられています。



このメニューからコマンドを指定することができる
ウォッチウィンドウ ここに刻々と変化する変数の内容が表示される
ウォッチンウンドウに表示する変数名を設定している
ソースコードが表示されるウィンドウ
SYMDEB コマンドも使うことができる
File
View
Search
Run
Watch
Options
Language
Calls
Help
F8=Trace F5=Go
attrib.c

```
0) *main.argv,s  :  "FOO"
1) *main.p,c  :
2) main.mask,x  :  0001
3) main.attr,x  :  0001

        Enter expression to be added to Watch Window:

        doserrno

           }
           *attr = ((reg.x.cx & ~mask) ¦ (mask & *attr));
           if (*attr != reg.x.cx) {
                   reg.x.ax = 0x4301;
                   reg.x.cx = *attr;
                   intdos(&reg, &reg);

Microsoft (R) CodeView (R)  Version 2.2
(C) Copyright Microsoft Corp. 1986-1988.  All rights reserved.
>
```

図 6.17　CODEVIEW におけるウォッチウィンドウとウォッチポイントの設定

CODEVIEW を利用するためには、SYMDEB の場合のようにシンボルファイルを作成するのではなく、シンボルやソースコードの情報を埋め込んだ実行ファイルを作成します。たとえば MS-C の場合、次の図 6.18 のようになります。

---

* その代わり機種間の互換性はなく、各機種専用の CODEVIEW を入手しなければならない。

```
A>TYPE CHMOD.MK↵
SRCS = chmod.c attrib.c
HDRS = attrib.h
OBJS = chmod.obj attrib.obj
DEFS = /DDEBUG
CONF =

CC = cl
MODEL = S
CFLAGS = /A$(MODEL) $(CONF) /DMSC=5 $(DEFS)

chmod.obj: chmod.c attrib.h
        $(CC) $(CFLAGS) /c /o chmod.obj chmod.c

attrib.obj: attrib.c attrib.h
        $(CC) $(CFLAGS) /c /o attrib.obj attrib.c

chmod.exe: $(OBJS)
        $(CC) $(CFLAGS) /o chmod.exe $(OBJS)

A>MAKE DEFS=/Dregister= CONF=/Zi CHMOD.MK↵
Microsoft (R) Program Maintenance Utility  Version 4.07
Copyright (C) Microsoft Corp 1984-1988.  All rights reserved.


MAKE : warnning U4000: 'chmod.obj' : target does not exist
  cl /AS /Zi /DMSC=5 /Dregister= /c /o chmod.obj chmod.c
Microsoft (R) C Optimizing Compiler Version 5.10
Copyright (c) Microsoft Corp 1984, 1985, 1986, 1987, 1988. All rights reserved.

chmod.c
```

MAKEファイルの内容を表示する

設定されているMAKEファイルの内容

レジスタ変数はワッチできないのでregisterを再定義して無効にする

MAKEを実行する

CODEVIEW用のオブジェクトを出力させるオプション

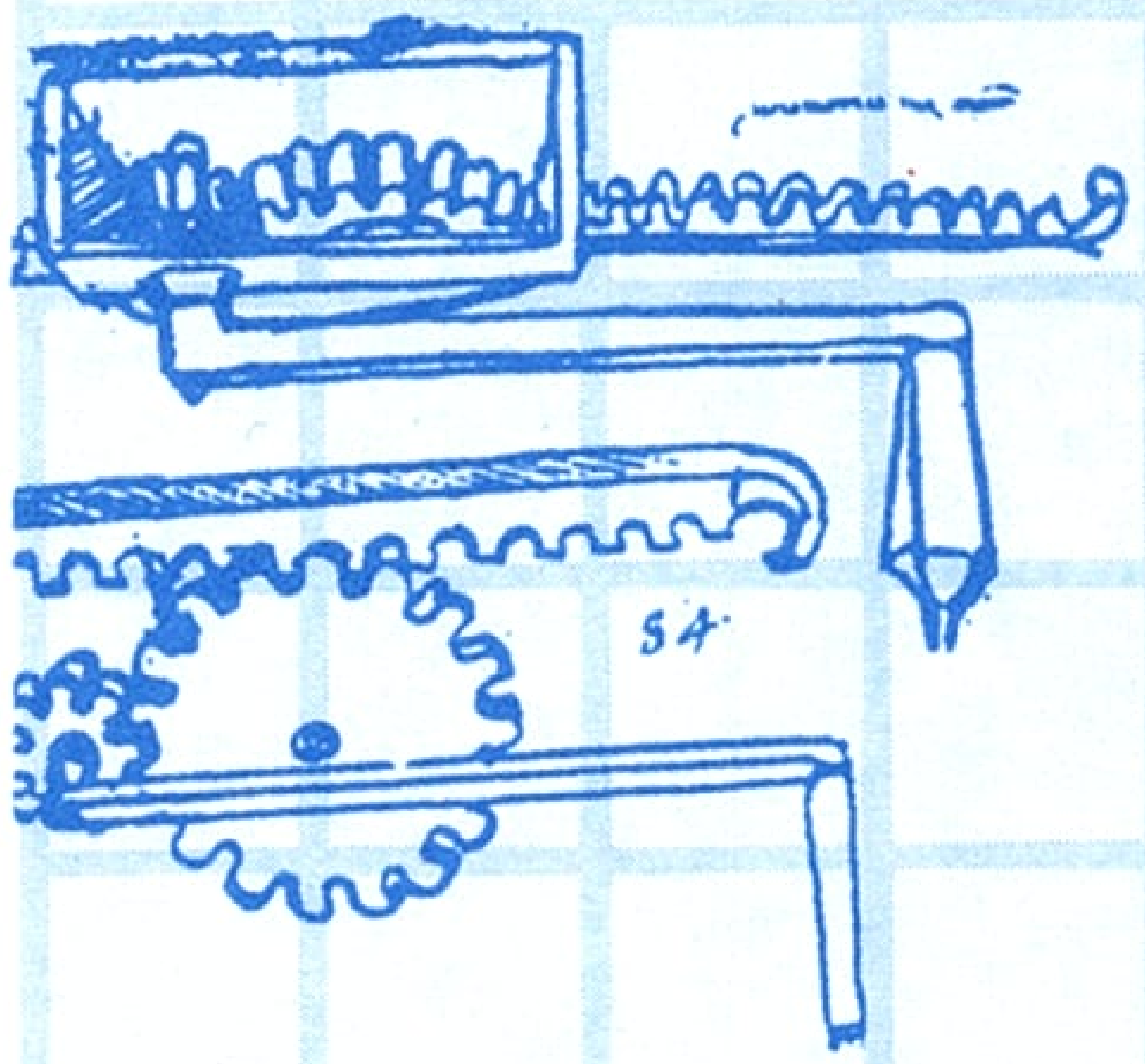
MAKEファイルの設定に従って実行されるコマンド（エコーバックされている）

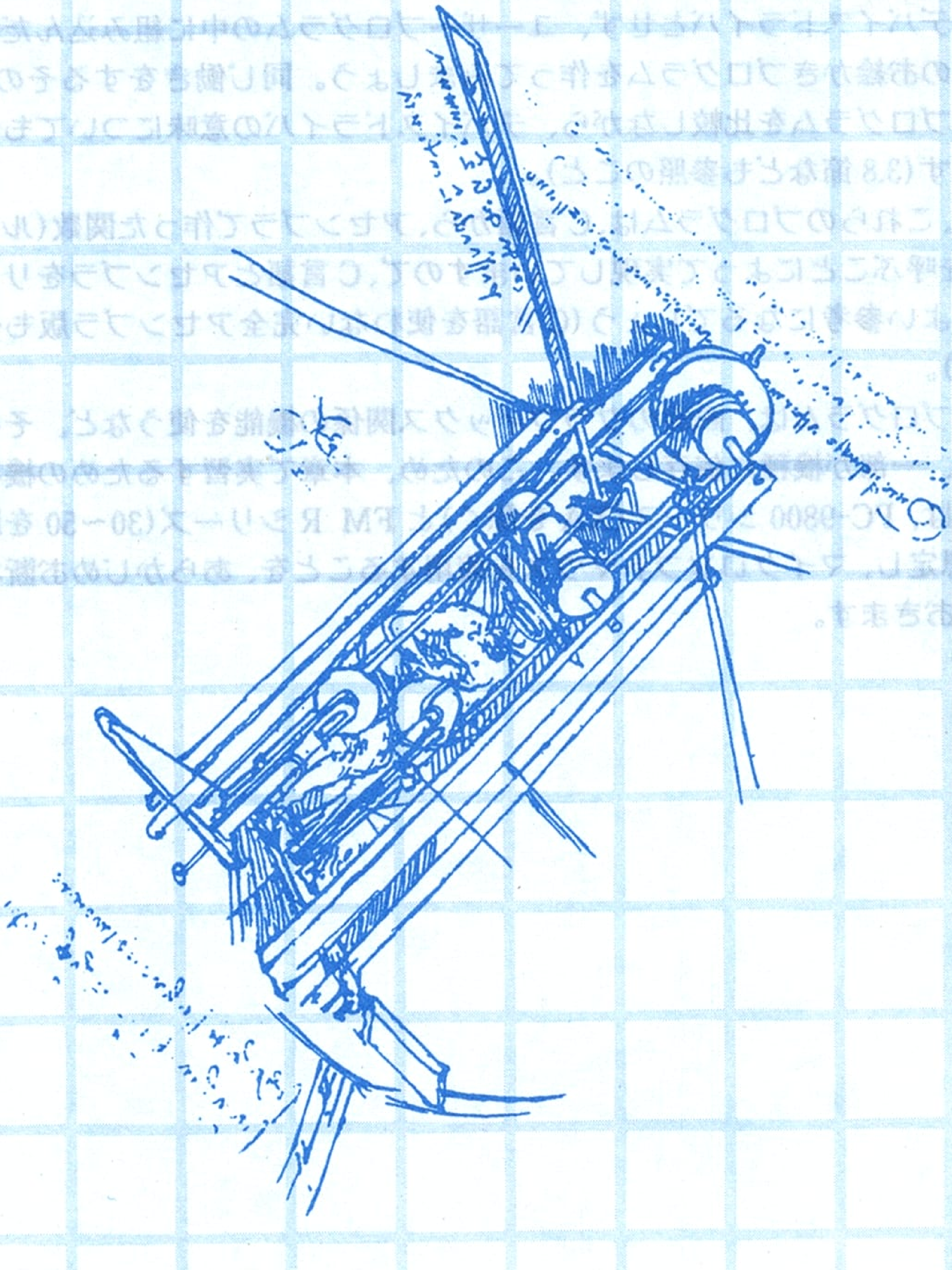
（以下略）

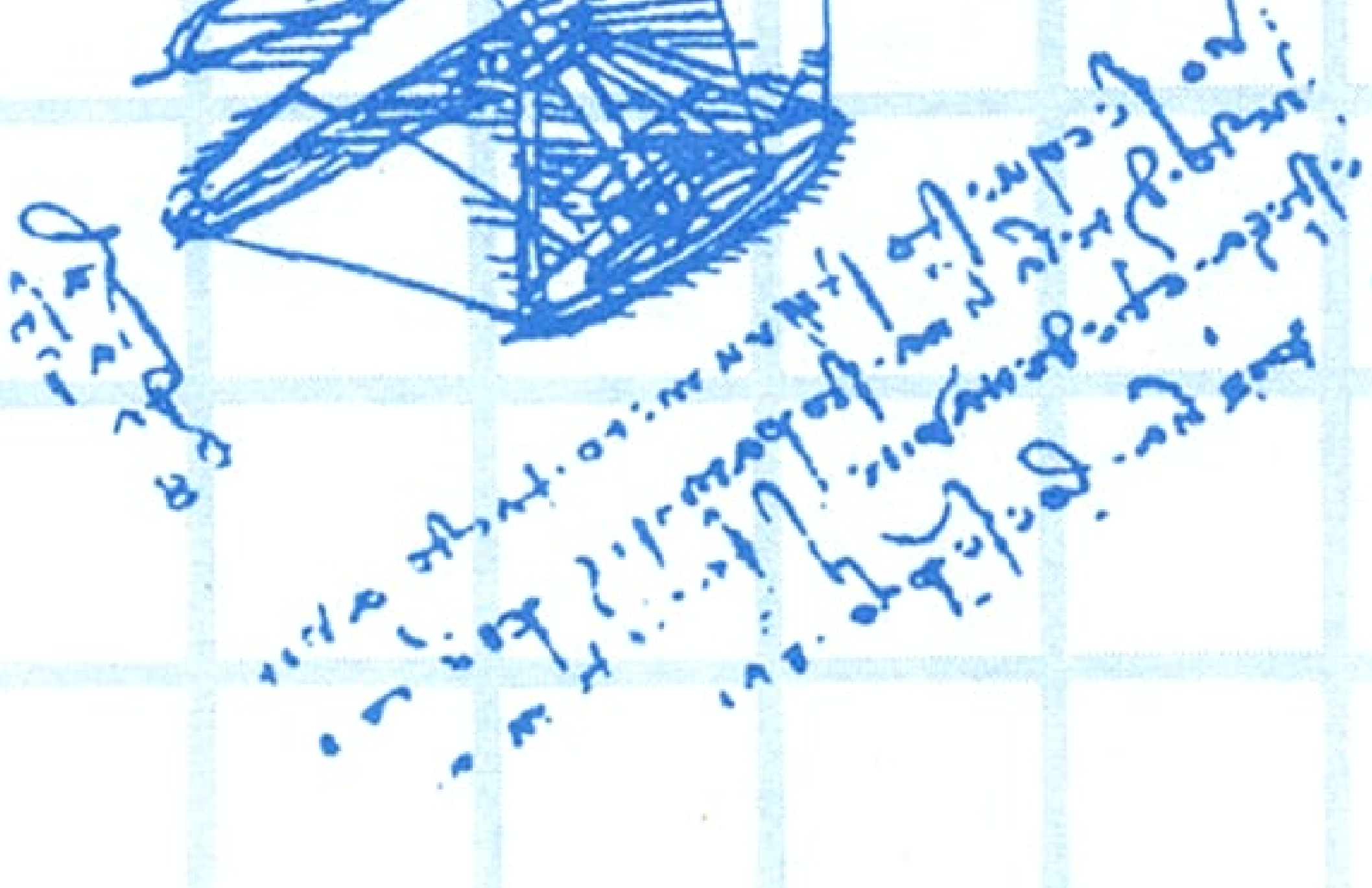
図6.18　CODEVIEWを利用するためのコンパイル手順

7章

デバイスドライバの作成とマウスの応用







本書の最後に、これまで解説してきたMS-DOSの知識をもとにして、マウスを使ってフリーハンドの線を描く「お絵かきプログラム」を作成してみましょう。このプログラムには、グラフィック画面上の指定した座標にドットを打つデバイスドライバを作成し、それをユーザープログラムからアクセスすることにより実現します。このプログラムは、キャラクタ型のデバイスドライバの作成やその使い方のよい参考になるでしょう。

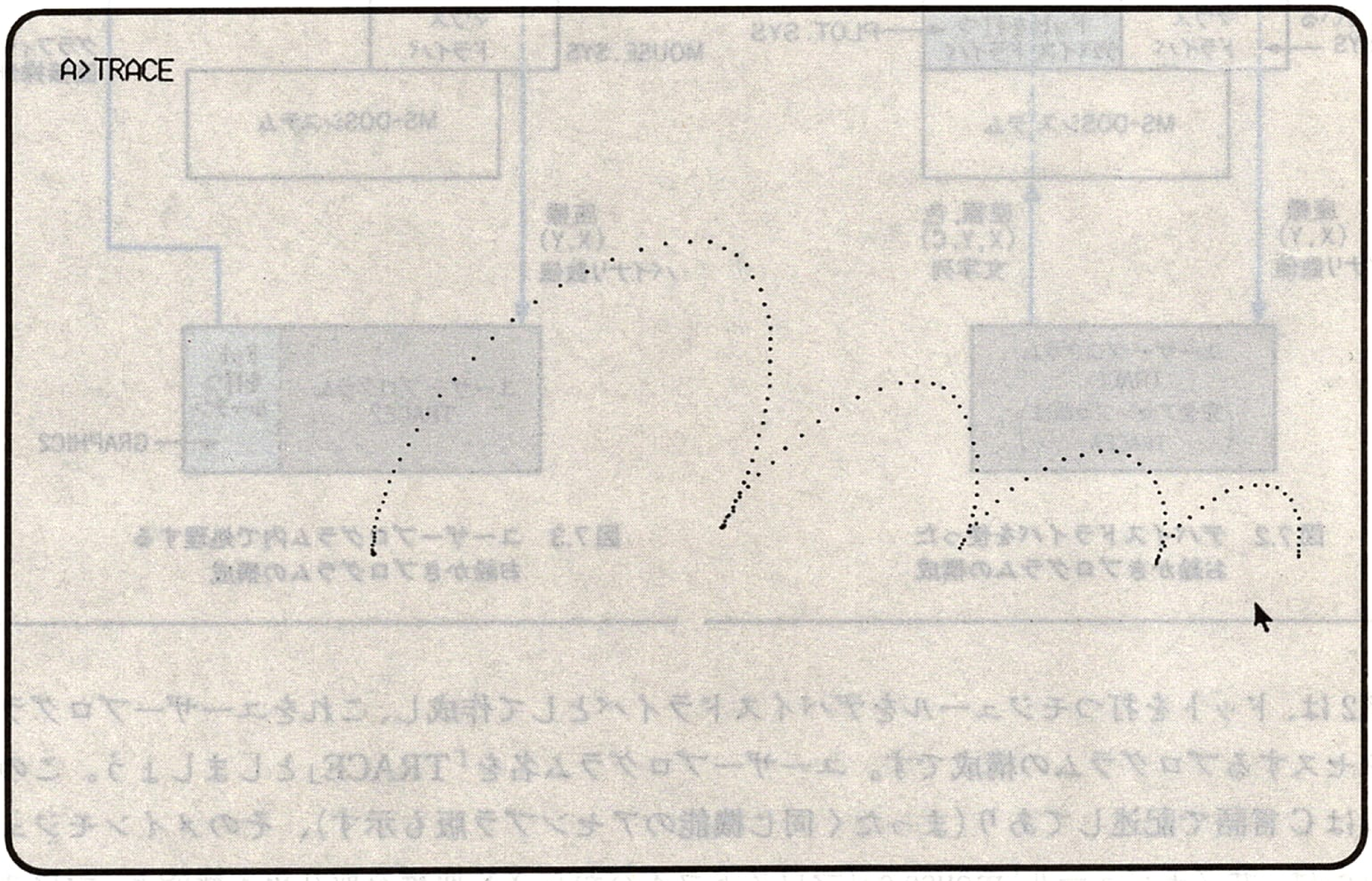
また、デバイスドライバを理解するための参考に、さきのドットを打つ部分をデバイスドライバとせず、ユーザープログラムの中に組み込んだ同じ機能のお絵かきプログラムを作ってみましょう。同じ働きをするその2種類のプログラムを比較しながら、デバイスドライバの意味についても解説します（3.8節なども参照のこと）。

また、これらのプログラムは、C言語から、アセンブラで作った関数（ルーチン）を呼ぶことによって実現していますので、C言語とアセンブラをリンクするよい参考になるでしょう（C言語を使わない完全アセンブラ版も作成する）。

なおプログラムは、画面のグラフィックス関係の機能を使うなど、その性質上、一部が機種に依存します。このため、本章で実習するための機種としては、PC-9800シリーズ（XAを除く）とFM Rシリーズ（30～50を除く）を想定し、マイクロソフトマウスを使用することを、あらかじめお断わりしておきます。

7.1 作成する「お絵かきプログラム」の構成

本章では、マウスを使って、フリーハンドの線を描くお絵かきプログラムを作成します。これは、**図 7.1** に示すように、マウスの移動に従って画面上のマウスカーソルの軌跡を描くものです。



```
A>TRACE
```

図 7.1　本章で作成するお絵かきプログラムで描いた画面例

このプログラムを実現するには、マウスから出力される XY 座標に対して、ドットを打つモジュールが必要です。プログラムの構成としては、このモジュールをデバイスドライバとするか、あるいはユーザープログラム内のただのルーチンとするかの 2 つの方法が考えられますが、本章では、この両方の形式のプログラムを作成してみましょう。これをもとに、デバイスドライバの作成法や、2 つの形式のプログラムの意味の違いなどを解説します。

まず、この 2 種類のプログラムの構成を**図 7.2**、**図 7.3** に示します。

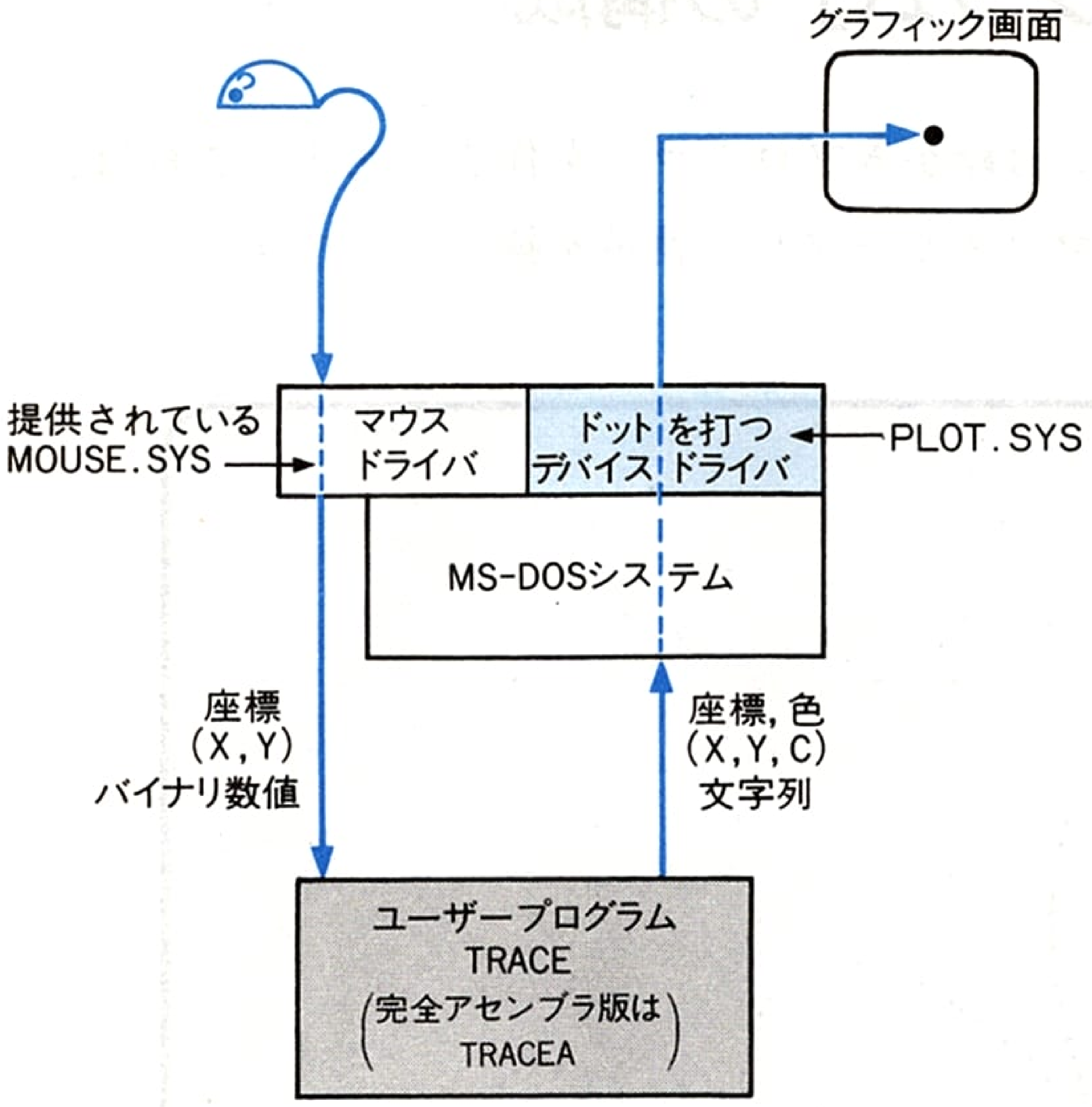

グラフィック画面
提供されている
MOUSE.SYS
マウス
ドライバ
ドットを打つ
デバイスドライバ
PLOT.SYS
MS-DOSシステム
座標
(X,Y)
バイナリ数値
座標,色
(X,Y,C)
文字列
ユーザープログラム
TRACE
(完全アセンブラ版は
TRACEA)

図7.2　デバイスドライバを使った
お絵かきプログラムの構成

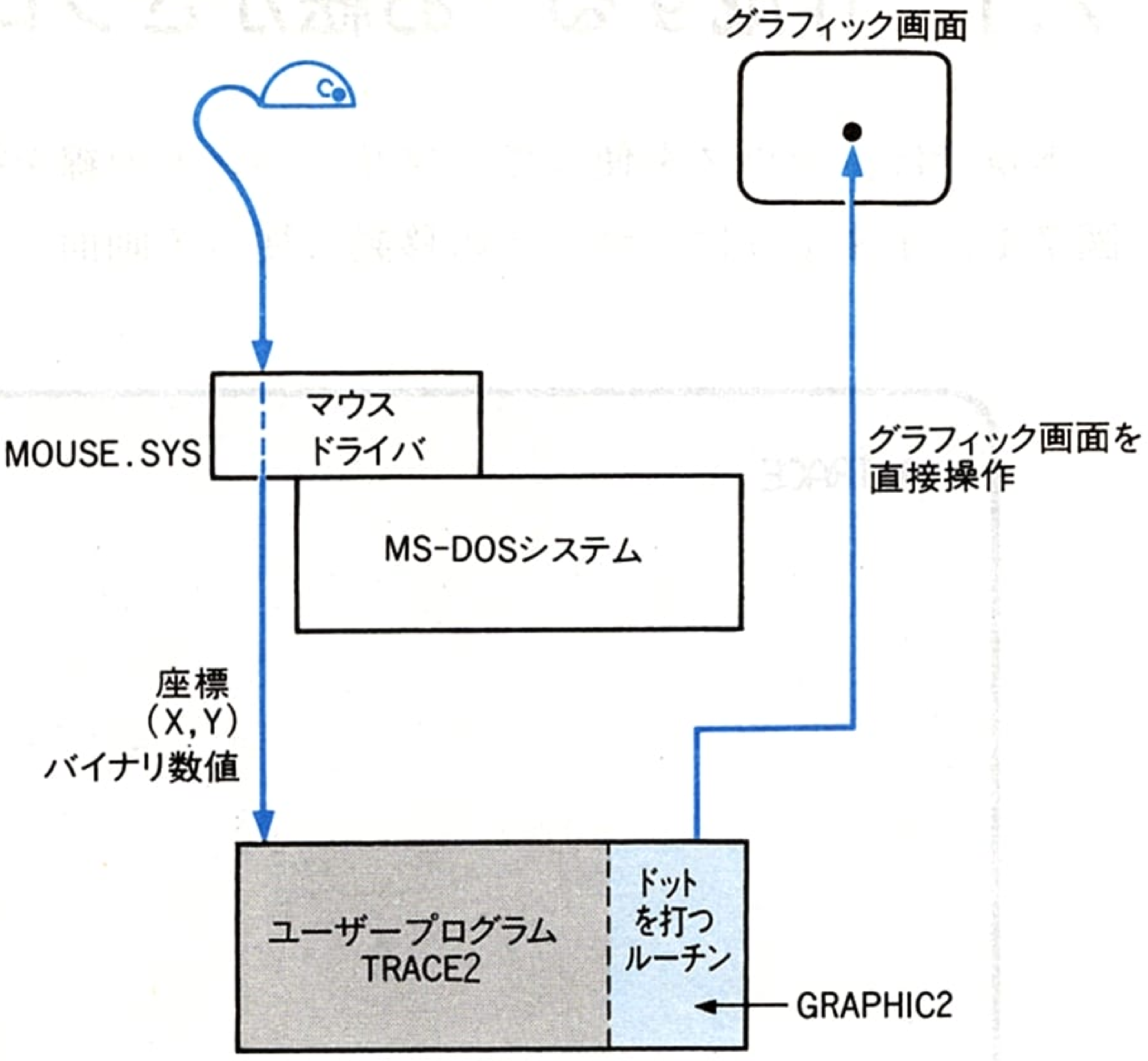

グラフィック画面
MOUSE.SYS
マウス
ドライバ
MS-DOSシステム
グラフィック画面を
直接操作
座標
(X,Y)
バイナリ数値
ユーザープログラム
TRACE2
ドット
を打つ
ルーチン
GRAPHIC2

図7.3　ユーザープログラム内で処理する
お絵かきプログラムの構成

図7.2は、ドットを打つモジュールをデバイスドライバとして作成し、これをユーザープログラムからアクセスするプログラムの構成です。ユーザープログラム名を「TRACE」としましょう。このプログラムはC言語で記述してあり(まったく同じ機能のアセンブラ版も示す)、そのメインモジュール「trace.c」は、サブモジュール「mouse.c」(マウスドライバからXY座標を取り出す機能と、マウスカーソルのON/OFFの機能を持つ)を呼ぶことによってマウスの座標パラメータを取り出し、そのパラメータを、サブモジュール「graphic.c」に渡します。このモジュールは、ドットを打つデバイスドライバ「PLT」を呼んでグラフィック画面に「書き込む」ようになっています。以上のようにして、このお絵かきプログラムを実現しています。なお、このデバイスドライバは、「DEVPLOT.ASM」としてアセンブラで記述します。

また、C言語で書かれたこのプログラムのメインモジュールを、アセンブラで書き直し、完全アセンブラ版としたプログラムも参考として作成します。アセンブラ版のプログラム名は「TRACEA」としましょう。

図7.3は、ドットを打つモジュールを、ユーザープログラム上のただのルーチンとして作成し、これをメインモジュールから利用するプログラムの構成です。このプログラム名を「TRACE2」としましょう。メインモジュール「trace.c」と「mouse.c」は、TRACEと同じもので、マウスドライバからのXY座標の取り出しと、マウスカーソルのON/OFFを行う部分を含み、ドットを打つモジュール

「GRAPHIC2.ASM」を呼ぶことにより機能します。このプログラムもC言語で記述します。なお、ドットを打つモジュールはアセンブラで記述してあり、関数モジュールとして作成されています。

マウスドライバ

使用するマウスドライバは、どちらのプログラムも、マイクロソフトマウス用に提供されている標準マウスドライバを使います。マウスドライバを自作するのは、けっこうたいへんですので、メーカーが提供するマウスドライバがあれば、それを利用すべきでしょう。PC-9800シリーズ用のマウスドライバは、マウスを利用するアプリケーションソフトには「MOUSE.SYS」というファイル名で付属していますのでそれを使います。また、FM Rシリーズでは、BIOSがマウスをサポートしていますので、それを利用します。ただし、いずれの場合も、「mouse.c」の書き換えが必要です。

7.2　デバイスドライバを利用したお絵かきプログラムの作成

とりあえず、このお絵かきプログラムの作成作業を始めましょう。解説は、作業とともに行います。この作業は、大きく分けてデバイスドライバの作成とユーザープログラムの作成とに分かれ、最初にデバイスドライバを作成し、その動作テストを行ったあとで、デバイスドライバをアクセスするユーザープログラムを作成します。

7.2.1　ドットを打つデバイスドライバの作成

解説はあとまわしにして、まずはデバイスドライバ版のお絵かきプログラムTRACEで使用する、指定した座標にドットを打つデバイスドライバPLOT.SYSを作成しましょう。デバイスドライバについては3章で簡単に解説しましたが、ここでは実際に、キャラクタ型のデバイスドライバを作成し、それをユーザープログラムからアクセスしてみます。

MS-DOSマシンの多くは、その本体にはグラフィック機能を持っていますが、その仕様が機種ごとに異なるため、MS-DOSではサポートされていません。このプログラムは、そのグラフィック画面を新たなデバイスと考えて、グラフィック画面を操作するデバイスドライバを作成し、これをシステムに追加してユーザープログラムからアクセスしようというものです。それによって、ユーザーはシステムを介してグラフィック画面をデバイスとして操作することができます。その意味するところについてはあとで考えることにして、まずこのデバイスドライバを作成してみましょう。

■ 作成するデバイスドライバの仕様

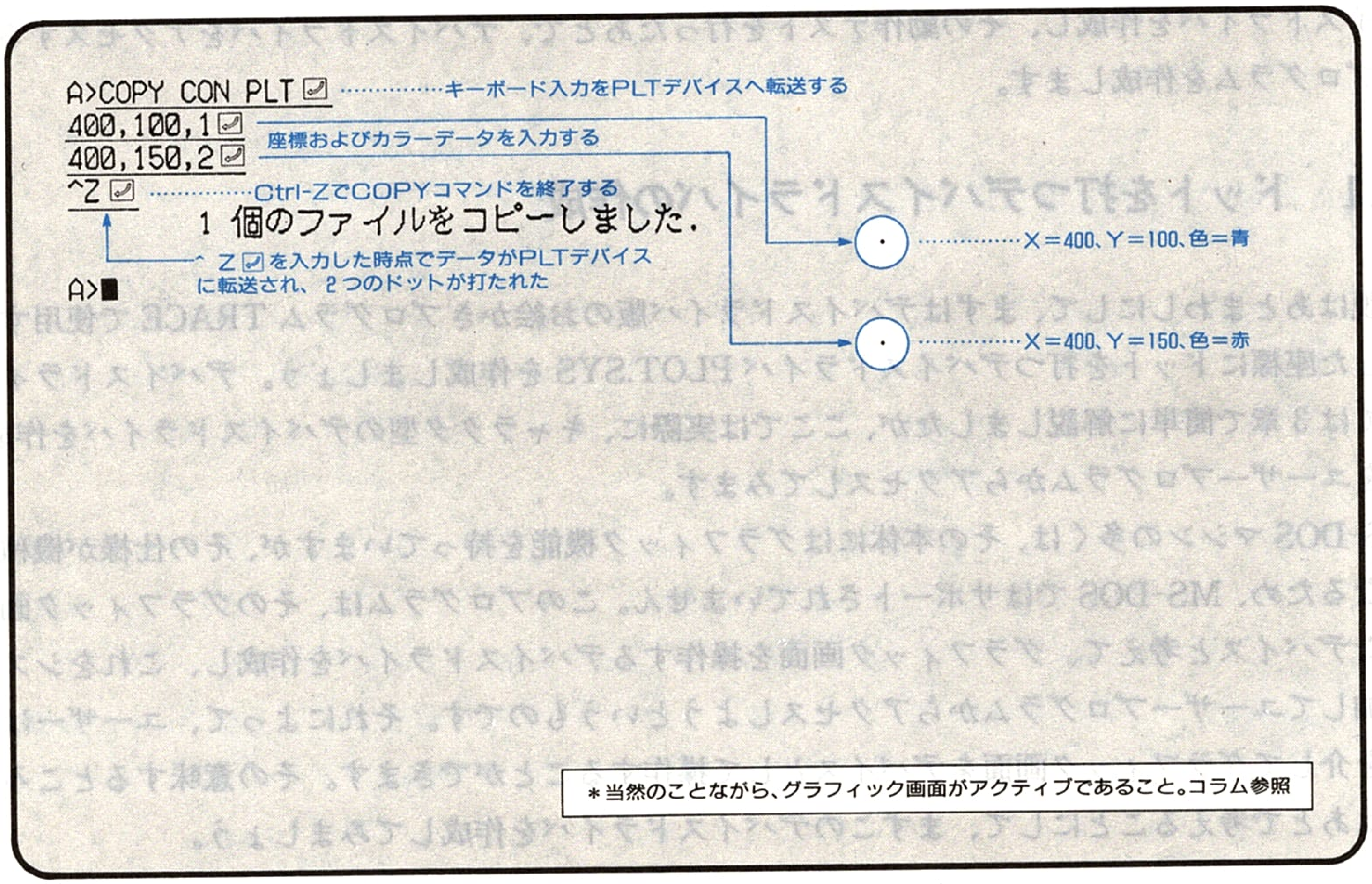
作成するプログラムは、グラフィック画面の指定した座標にドットを打つデバイスドライバです。このデバイス名を「PLT」という名前にしましょう。なお、最終的に作成するデバイスドライバの名前(デバイスドライバのプログラムファイルの名前)は「PLOT.SYS」ですが、デバイス名は「PLT」であることに注意してください。コンソールが「CON」、プリンタが「PRN」であるように、このドライバは「PLT」としてMS-DOSに認識されます。

グラフィック画面にドットを打つには、このPLTというキャラクタ型デバイスに、ASCII文字列によるパラメータ、「X座標」「Y座標」「カラーコード」をカンマやスペース、キャリッジリターンで区切って書き込むことにより、指定した座標(x,y)に指定した色の点が打たれます。

キャラクタ型デバイスに入力する座標値などのパラメータは、ASCII文字列でなくバイナリ形式で送る方が効率はよいのですが、キャラクタ型デバイスにバイナリ値を送った場合は、MS-DOSのシステムが、バイナリ値の一部(Ctrl-Zなど)をコントロールコードと解釈し、正常に動作しなくなるような不都合が生じます。このため、わざわざテキスト形式の文字列でパラメータを指定しています。

このデバイスドライバの働きを別のことばで表現すれば、「XY座標と色のパラメータを書き込んだテキストファイルを、このデバイスにCOPYコマンドでコピー(転送)することにより、グラフィック画面に指定したドットが打たれる」ということになります。具体例を図7.4に示します。

なお、指定できるドットの座標の範囲とカラーコードは図7.5のとおりです。

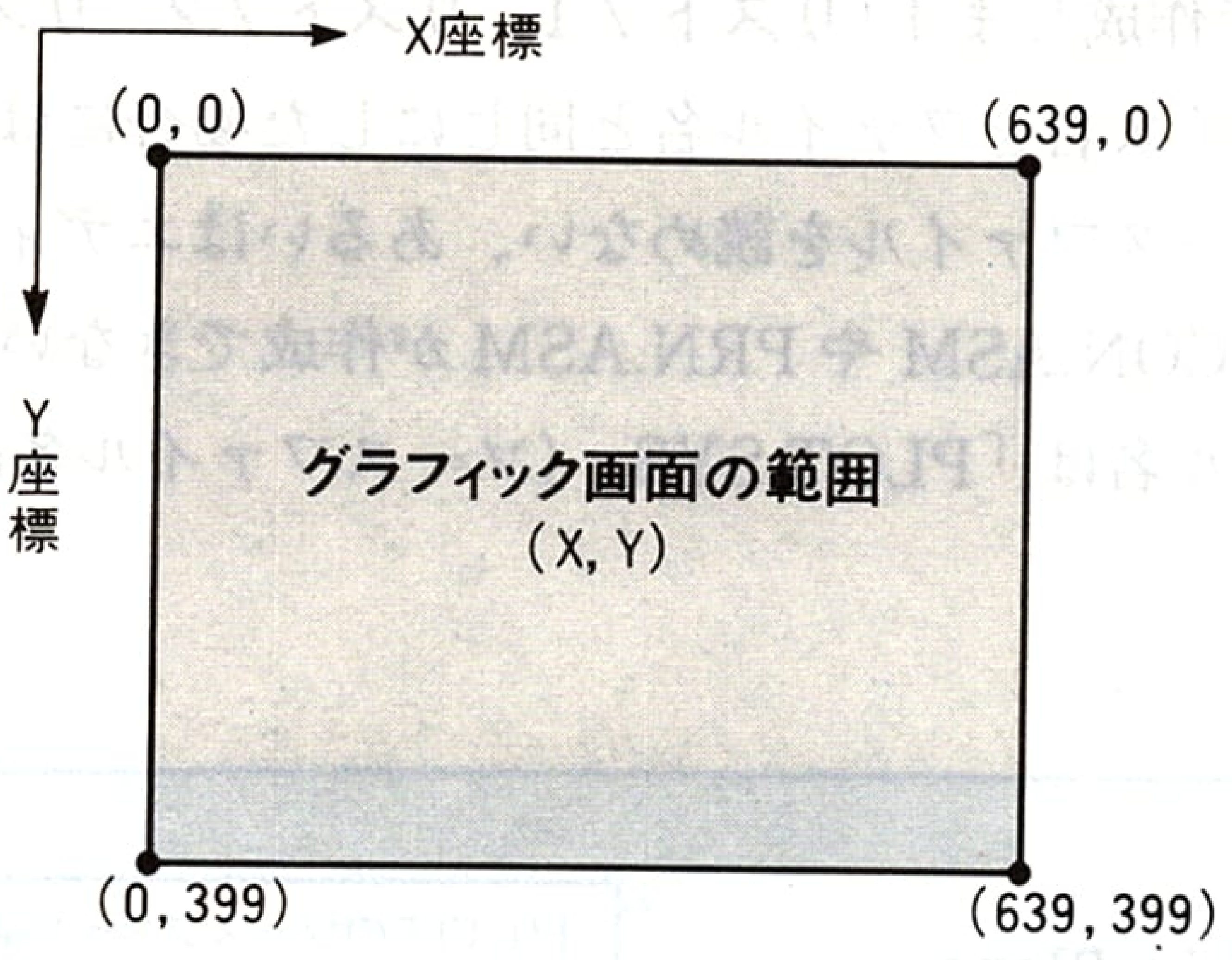


```
A>COPY CON PLT↵
400,100,1↵
400,150,2↵
^Z↵
        1 個のファイルをコピーしました.

A>■
```

キーボード入力をPLTデバイスへ転送する
座標およびカラーデータを入力する
Ctrl-ZでCOPYコマンドを終了する
^Z↵を入力した時点でデータがPLTデバイスに転送され、2つのドットが打たれた
X＝400、Y＝100、色＝青
X＝400、Y＝150、色＝赤

＊当然のことながら、グラフィック画面がアクティブであること。コラム参照

図7.4　COPYコマンドでPLTデバイスをアクセスする例



カラーコード	色
0	黒
1	青
2	赤
3	紫
4	緑
5	水色
6	黄
7	白

カラーコード

図 7.5　指定できるドットの座標の範囲とカラーコード

グラフィック画面のON／OFFについて

グラフィック画面を操作するためには、いずれの場合も、グラフィック画面が「有効」(アクティブ)になっていなければなりません。以下に、そのためのプログラムを参考までに示します。なお、グラフィック画面を無効にするプログラムも示しておきます。

```
INCLUDE CONFIG.INC

CODE    SEGMENT
        ASSUME  CS:CODE
        ORG     100H
START:
IF MACHINE EQ PC9801
        MOV     AH,40H       PC-9800シリーズ用
        INT     18H          ルーチン
ENDIF ; PC9801
IF MACHINE EQ FMR60
        MOV     AL,00000101B
        MOV     DX,0400H     FM R-60/70
        OUT     DX,AL        用ルーチン
ENDIF ; FMR60
        MOV     AX,4C00H
        INT     21H
CODE    ENDS
        END     START
```

グラフィック画面を有効にするプログラム　GRON.ASM

```
INCLUDE CONFIG.INC

CODE    SEGMENT
        ASSUME  CS:CODE
        ORG     100H
START:
IF MACHINE EQ PC9801
        MOV     AH,41H       PC-9800シリーズ用
        INT     18H          ルーチン
ENDIF ; PC9801
IF MACHINE EQ FMR60
        MOV     AL,00000001B
        MOV     DX,0400H     FM R-60/70
        OUT     DX,AL        用ルーチン
ENDIF ; FMR60
        MOV     AX,4C00H
        INT     21H
CODE    ENDS
        END     START
```

グラフィック画面を無効にするプログラム　GROFF.ASM

■ デバイスドライバPLOTのソースファイルの作成

デバイスドライバPLOTはアセンブラで作成します（**リスト7.1、リスト7.2、リスト7.3、リスト7.4**）。デバイス名は「PLT」ですが、デバイス名をファイル名と同じにした場合には、そのデバイスがシステムに登録されていると、同名のソースファイルを読めない、あるいはエディットできないなど、具合いが悪いことになってしまうので（CON.ASMやPRN.ASMが作成できないのと同じ）、それを避けるために、作成するデバイスファイル名は「PLOT.SYS」（ソースファイル名は「DEVPLOT.ASM」）ですので注意してください。

```
;; DEVPLOT.ASM  Device Driver
;;              For Plotting Graphic Plane

INCLUDE CONFIG.INC
INCLUDE DEVICE.INC

PUSHM   MACRO   RLIST
        IRP     REG,<RLIST>
        PUSH    REG
        ENDM
        ENDM

POPM    MACRO   RLIST
        IRP     REG,<RLIST>
        POP     REG
        ENDM
        ENDM

CSEG    SEGMENT
        ASSUME  CS:CSEG

        ORG     0

        DEVHDR  -1,8000H,STRENT,INTRENT,DEVNAME

CMDTBL  DW      INIT
        DW      EXIT
        DW      EXIT
        DW      CMDERR
        DW      EXIT
        DW      EXIT
        DW      EXIT
        DW      EXIT
        DW      OUTPUT
        DW      OUTPUT
        DW      EXIT
        DW      EXIT
        DW      EXIT
```

PLOTのソースファイルは、このメインルーチン「DEVPLOT.ASM」のほかに、INCLUDEされる（自動的に結合される）3つのソースファイル「CONFIG.ASM」「DEVICE.INC」「GRAPH.ASM」の合計4つから成る。

ORG 0 ……デバイスドライバには「ORG 100$_H$」を付けないことに注意

DEVHDR ……デバイスヘッダ。これだけはプログラムの先頭に固定する

DEVNAME ― デバイス名

INTRENT ― 割り込みルーチンへのポインタ

STRENT ― ストラテジルーチンへのポインタ

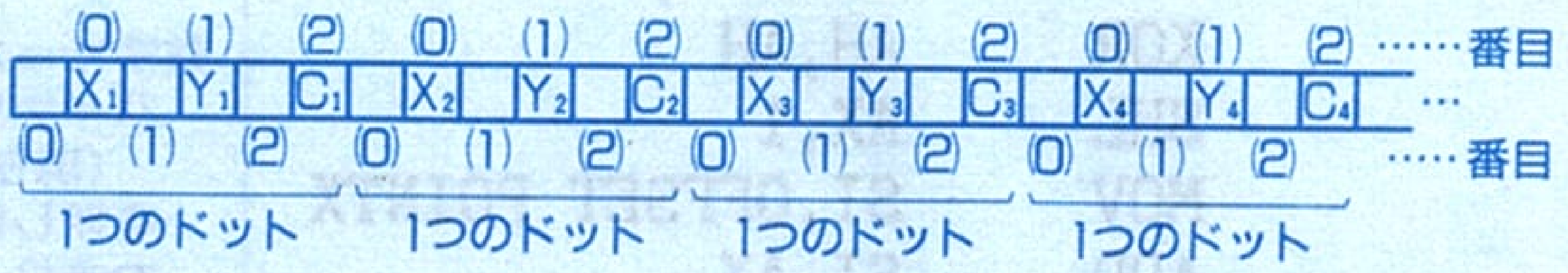
8000H ― 標準デバイスを置き換えないタイプのキャラクタ型デバイスの場合は8000$_H$をセットする

-1 ― 次のデバイスドライバへのポインタ。ここでは1つしかないので−1（FFFF:FFFF$_H$）に

CMDTBL ― I/Oリクエストコマンド用のジャンプテーブル。このプログラムでは、コマンドコード0のINITと、8および9のOUTPUTのみサポートしている

— リスト7.1 —（次ページ以下に続く）

```
TBLLEN  EQU     ($ - CMDTBL) / 2
PACKADR DD      ? …………コマンドパケットのアドレスを格納するバッファ

CMDCNT  DB      0 …何番目の値を入力中かを示す→
BLKCNT  DB      -1 …何番目の区切りかを示す →
POINTX  DW      0 …X座標値を格納する
POINTY  DW      0 …Y座標を格納する
COLOR   DW      0 …カラーコード(C)を格納する
```



(0) (1) (2) (0) (1) (2) (0) (1) (2) (0) (1) (2) ……番目
X1 Y1 C1 X2 Y2 C2 X3 Y3 C3 X4 Y4 C4 …
(0) (1) (2) (0) (1) (2) (0) (1) (2) (0) (1) (2) ……番目
1つのドット
1つのドット
1つのドット
1つのドット

```
        PUBLIC  STRENT
STRENT  PROC    FAR
        MOV     WORD PTR PACKADR,BX      } コマンドパケットのアドレスを格納してすぐ戻る
        MOV     WORD PTR PACKADR+2,ES    }
        RET
STRENT  ENDP

        PUBLIC  INTRENT
INTRENT PROC    FAR
        PUSHM   <AX,BX,CX,DX,SI,DI,BP,DS,ES> …………割り込みルーチンでは、すべてのレジスタを
                                                 退避しなければならない
        LDS     BX,PACKADR …………コマンドパケットのアドレスを取り出す
        MOV     AL,[BX].COMMAND …………コマンドコードを得る
        CMP     AL,TBLLEN    } コマンドコードは最大2まで(MS-DOSバージョン3.1では16番までの17個。
        JAE     CMDERR       } バージョン3.3では、23番までの20個)
        XOR     AH,AH
        SHL     AX,1             } I/Oリクエストコマンドのジャンプテーブルを参照し、コマンド
        MOV     SI,AX            } コードに対する各処理ルーチン(各コマンド)へ分岐する
        JMP     CS:CMDTBL[SI]    }

EXIT:   MOV     AX,0100H …………正常終了のコードを返す(DONEビットを1にして返す)
ERREXT: LDS     BX,CS:PACKADR …………ステータスをコマンドパケットに入れて返す
        MOV     DS:[BX].STATUS,AX
        POPM    <ES,DS,BP,DI,SI,DX,CX,BX,AX> …………割り込みルーチンの先頭で退避した
                                                 レジスタを復帰する
        RET
INTRENT ENDP

CMDERR  PROC    NEAR            }
        MOV     AX,8103H        } コマンドコードの値が正常でなかった場合、エラーコードを返す
        JMP     SHORT ERREXT    }
CMDERR  ENDP                    }

        PUBLIC  OUTPUT
OUTPUT  PROC    NEAR
        MOV     CX,[BX].COUNT …………転送バイト数を得る
        LES     SI,[BX].TRANS …………転送バイト数が0なら終了する
OUTLOP: MOV     BL,ES:[SI] …………1文字取り出し、数字であるかどうかを調べる
        INC     SI
        CMP     BL,'0'
        JB      NDIGIT
        CMP     BL,'9'
        JA      NDIGIT
```

```
DIGIT:  SUB     BL,’0’
        XOR     BH,BH
        MOV     AL,CMDCNT
        MOV     BLKCNT,AL
        XOR     AH,AH
        SHL     AX,1
        MOV     SI,OFFSET POINTX
        ADD     SI,AX
        MOV     AX,10
        MUL     WORD PTR CS:[SI]
        ADD     AX,BX
        MOV     CS:[SI],AX
        JMP     SHORT NEXTCH
```

数字だった場合の処理
バイナリ数値に変換し、
（現在の値）×10＋（次の値）
を新しい値とする。
つまり、数字列を10進数とみなして
バイナリ数値に変換していく

```
NDIGIT: MOV     AL,CMDCNT
        CMP     AL,BLKCNT
        JNE     NEXTCH
        INC     CMDCNT
        CMP     AL,2
        JB      NEXTCH
        PUSH    CX
        PUSHM   <COLOR,POINTY,POINTX>
        CALL    PLOT
        ADD     SP,6
        POP     CX
```

数字でなかった場合の処理
座標値またはカラーコードを１つ読み終わったことにする
座標値２つとカラーコードを読み込んでいればドットを打つ

このようなプログラムのため、最後の座標（X、Y、C）の後ろに、空白やキャリッジリターンが必要となる

```
        XOR     AX,AX
        MOV     CMDCNT,AL
        MOV     POINTX,AX
        MOV     POINTY,AX
        MOV     COLOR,AX
```

作業領域を再初期化する

```
NEXTCH: LOOP    OUTLOP
        JMP     EXIT
OUTPUT  ENDP
CSEG    ENDS
```

……………転送バイト数が０になるまで繰り返す

```
INCLUDE GRAPH.ASM

CSEG    SEGMENT
        ASSUME  CS:CSEG
CODEEND LABEL   BYTE
        PUBLIC  INIT
INIT    PROC    NEAR
        LDS     BX,PACKADR
        MOV     AX,OFFSET CODEEND
        MOV     WORD PTR [BX].BRKADR,AX
        MOV     WORD PTR [BX].BRKADR+2,CS
        JMP     EXIT
INIT    ENDP
CSEG    ENDS
        END
```

初期化ルーチン
デバイスドライバのブレークアドレス（終了アドレス）を返す
初期化ルーチンをこのアドレスより後ろに置いてメモリを再利用することもできる

リスト 7.1　デバイスドライバ PLOT のソースプログラム DEVPLOT.ASM

```
;; CONFIG.INC

DEVNAME EQU     "PLT" ……………デバイス名

PC9801  EQU     1
FMR60   EQU     2

MACHINE EQU     PC9801 ……………機種依存部を条件アセンブルするための定義。FM R-60/70の
                          ときはここをMACHINE EQU FMR60とする
```

リスト 7.2　仕様定義プログラム CONFIG.INC

```
;; DEVICE.INC    Definitions of
;;               Device Request Packet

DEVHDR  MACRO    LINK,ATTRIB,STRENT,INTRENT,NAME
        LOCAL    DEVNAME
        DD       LINK
        DW       ATTRIB
        DW       STRENT
        DW       INTRENT
        DB       8 DUP (' ')
        ORG      $ - 8
DEVNAME DB       NAME
        ORG      OFFSET DEVNAME + 8
        ENDM

;;
;; Request Packete definitions
;;

;; Request Header

LEN     EQU      0
DEVNO   EQU      1
COMMAND EQU      2
STATUS  EQU      3
RSRVED  EQU      5

REQHDR  EQU      13

;; Initialize

UNITS   EQU      REQHDR
BRKADR  EQU      REQHDR+1
P2BPB   EQU      REQHDR+3
DRVS    EQU      REQHDR+7
```

DEVHDR MACRO … ENDM：デバイスヘッダを定義するためのマクロ

DB 8 DUP (' ') … ORG OFFSET DEVNAME + 8：マクロの引数NAMEが8文字でなかった場合に対処するため、ロケーションカウンタを操作する

LEN EQU 0 … DRVS EQU REQHDR+7：コマンドパケット内のパラメータのオフセットを定義する

— リスト 7.3 —（次ページに続く）

```
;; I/O request

MEDIA   EQU     REQHDR
TRANS   EQU     REQHDR+1
COUNT   EQU     REQHDR+5
SECTOR  EQU     REQHDR+7
VOLID   EQU     REQHDR+9
```

リスト 7.3　デバイスドライバ用定義プログラム DEVICE.INC

```
;; GRAPH.ASM      Graphic Plane Plot
;;                Procedures

IF MACHINE EQ PC9801

HSIZE   EQU     640
VSIZE   EQU     400       V-RAMのプレーン数と解像度を定義する
NPLANE  EQU     3

BGVRAM  SEGMENT AT 0A800H
BGVRAM  ENDS
RGVRAM  SEGMENT AT 0B000H
RGVRAM  ENDS                 V-RAMのセグメントを定義する
GGVRAM  SEGMENT AT 0B800H
GGVRAM  ENDS

ENDIF ; PC9801

IF MACHINE EQ FMR60

HSIZE   EQU     1120
VSIZE   EQU     750
NPLANE  EQU     4

LIMIT   EQU     400

GVRAM0  SEGMENT AT 0C000H
GVRAM0  ENDS
IF ((LIMIT * (HSIZE / 8)) AND 0FH) NE 0
ERROR
ENDIF
GVRAM1  SEGMENT AT 0C000H + ((LIMIT * (HSIZE / 8)) SHR 4)
GVRAM1  ENDS

ENDIF ; FMR60

CSEG    SEGMENT
        ASSUME  CS:CSEG
```

— リスト 7.4 — (次ページ以下に続く)

```
        PUBLIC  PLOT
PLOT    PROC    NEAR

PX      EQU     4
PY      EQU     6
COL     EQU     8

        PUSH    BP
        MOV     BP,SP
        PUSH    SI
        PUSH    DI
        PUSH    DS
        MOV     AX,[BP].PX
        MOV     BX,[BP].PY
        CALL    COMPADR
        MOV     BX,[BP].COL
        MOV     CX,NPLANE
IF MACHINE EQ PC9801
BITSET: MOV     DS,CS:[SI]
        INC     SI
        INC     SI
ENDIF ; PC9801
IF MACHINE EQ FMR60
        MOV     AX,1B
BITSET: MOV     SI,DX
        MOV     DX,0402H
        OUT     DX,AL
        SHL     AL,1
        XCHG    AL,AH
        MOV     DX,0404H
        OUT     DX,AL
        XCHG    AL,AH
        INC     AH
        MOV     DX,SI
ENDIF ; FMR60
        SHR     BL,1
        JNC     CLEAR
        OR      [DI],DL
        LOOP    BITSET
        JMP     RETURN
CLEAR:
        AND     [DI],DH
        LOOP    BITSET
RETURN:
        MOV     DS,[BP-6]
        MOV     DI,[BP-4]
        MOV     SI,[BP-2]
        MOV     SP,BP
        POP     BP
        RET
PLOT    ENDP
CSEG    ENDS
```

スタック上のパラメータのオフセット

座標値からV-RAM上のオフセットアドレスとビットアドレスを得る

カラーコードに応じてRGB各プレーンのビットをON／OFFする

ドットを打つ関数

入力
- POINTX　X座標
- POINTY　Y座標
- COLOR　カラーコード

出力
- 画面上の指定された座標に指定した色のドットが打たれる

```
CSEG     SEGMENT
         ASSUME   CS:CSEG
COMPADR PROC     NEAR
IF MACHINE EQ PC9801
         MOV      SI,OFFSET SEGTBL
ENDIF ; PC9801
IF MACHINE EQ FMR60
         MOV      SI,GVRAM0
         CMP      BX,LIMIT
         JB       CPADR
         SUB      BX,LIMIT
         MOV      SI,GVRAM1
CPADR:   MOV      DS,SI
ENDIF ; FMR60
         MOV      CX,AX
         MOV      AX,HSIZE / 8
         MUL      BX
         MOV      DI,AX
         MOV      AX,CX
         SHR      AX,1
         SHR      AX,1
         SHR      AX,1
         ADD      DI,AX
         AND      CL,111B
         MOV      DL,10000000B
         SHR      DL,CL
         MOV      DH,DL
         NOT      DH
         RET

IF MACHINE EQ PC9801
SEGTBL   DW       RGVRAM
         DW       GGVRAM
         DW       BGVRAM
ENDIF ; PC9801

COMPADR ENDP
CSEG     ENDS
```

V-RAM上の
オフセットアドレス
$\left(\frac{HSIZE}{8}\right)\times Y+\left(\frac{X}{8}\right)$
を計算する

ビットアドレスを計算する

V-RAMのセグメントアドレス

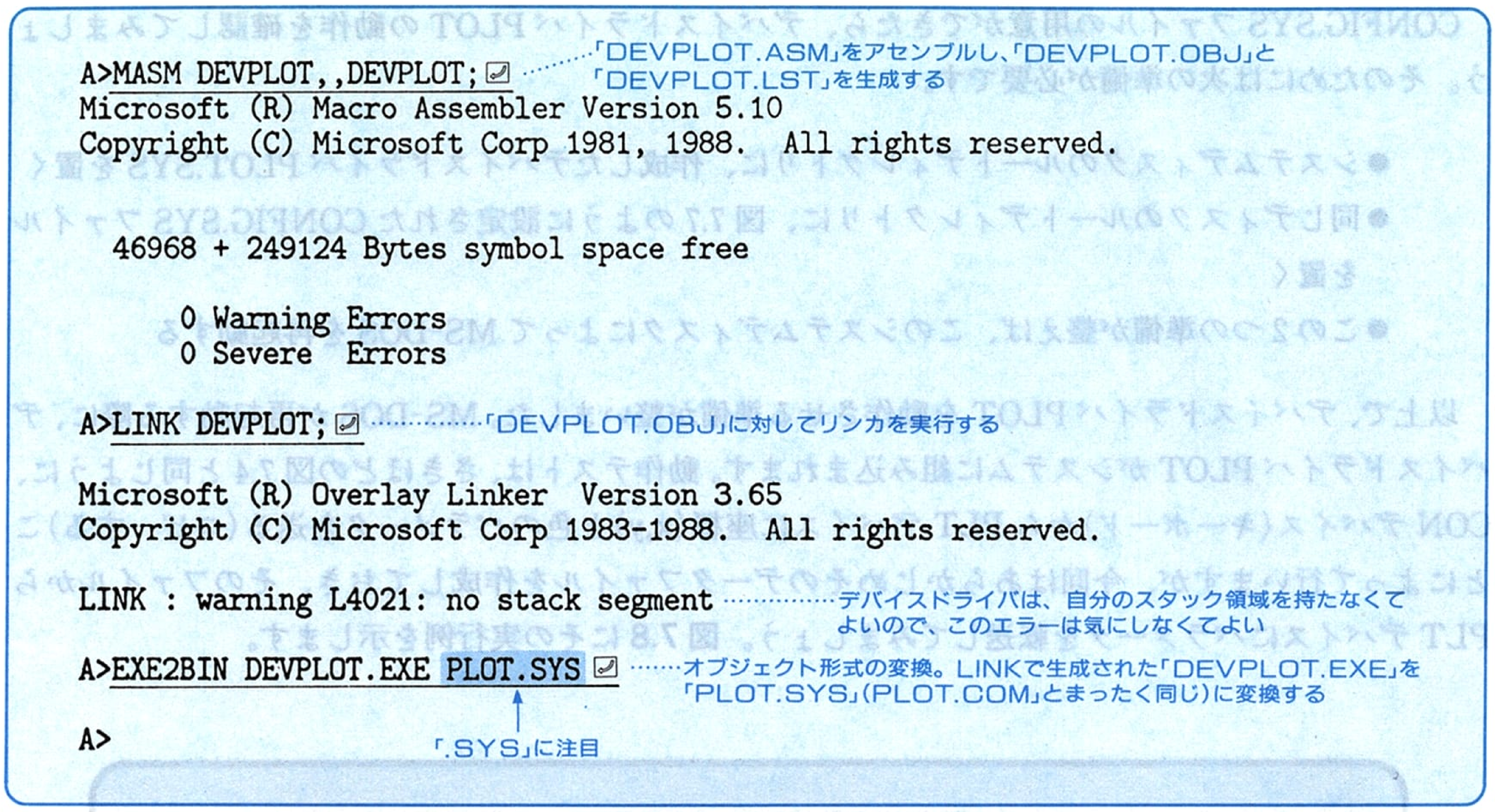
V-RAM上のオフセット
アドレスを計算する関数

入力 AX＝X座標
BX＝Y座標

出力 DI＝V-RAMオフセット
アドレス
DL＝ビットセット用マスク
DH＝ビットリセット用マスク

リスト7.4　プロットルーチンのソースプログラムGRAPH.ASM

ソースファイルがすべてできあがったら、アセンブルおよびリンク、それにオブジェクト形式の変換作業を行い、実行可能なデバイスドライバにします(図7.6参照)。この作業は、基本的にはCOMファイルを作るときの手順と同じですが、デバイスドライバ(つまりシステムの一部)であることを示すため、最終的なデバイスドライバのファイル名は、EXE2BIN実行時に拡張子を「.SYS」に指定して、PLOT.SYSとしておくとよいでしょう。



```
A>MASM DEVPLOT,,DEVPLOT;↵
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


  46968 + 249124 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

A>LINK DEVPLOT;↵

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

LINK : warning L4021: no stack segment

A>EXE2BIN DEVPLOT.EXE PLOT.SYS↵

A>
```

「DEVPLOT.ASM」をアセンブルし、「DEVPLOT.OBJ」と「DEVPLOT.LST」を生成する

「DEVPLOT.OBJ」に対してリンカを実行する

デバイスドライバは、自分のスタック領域を持たなくてよいので、このエラーは気にしなくてよい

オブジェクト形式の変換。LINKで生成された「DEVPLOT.EXE」を「PLOT.SYS」(PLOT.COMとまったく同じ)に変換する

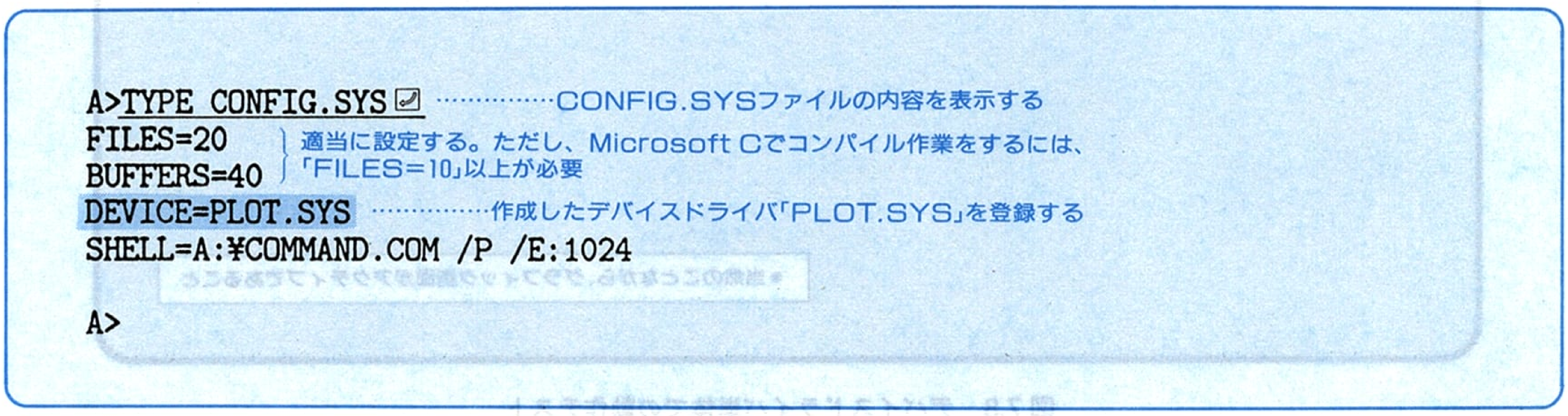
「.SYS」に注目

図 7.6　ソースプログラムから実行可能なデバイスドライバ PLOT.SYS を作成する

以上の作業で、画面にドットを打つデバイスドライバ PLOT ができあがりました。

■ デバイスドライバ PLOT の動作確認

作成されたデバイスドライバを、MS-DOS システムに組み込むには、CONFIG.SYS ファイルに図 7.7 のように登録しておかなければなりません。



```
A>TYPE CONFIG.SYS↵
FILES=20
BUFFERS=40
DEVICE=PLOT.SYS
SHELL=A:¥COMMAND.COM /P /E:1024

A>
```

CONFIG.SYSファイルの内容を表示する

適当に設定する。ただし、Microsoft Cでコンパイル作業をするには、「FILES=10」以上が必要

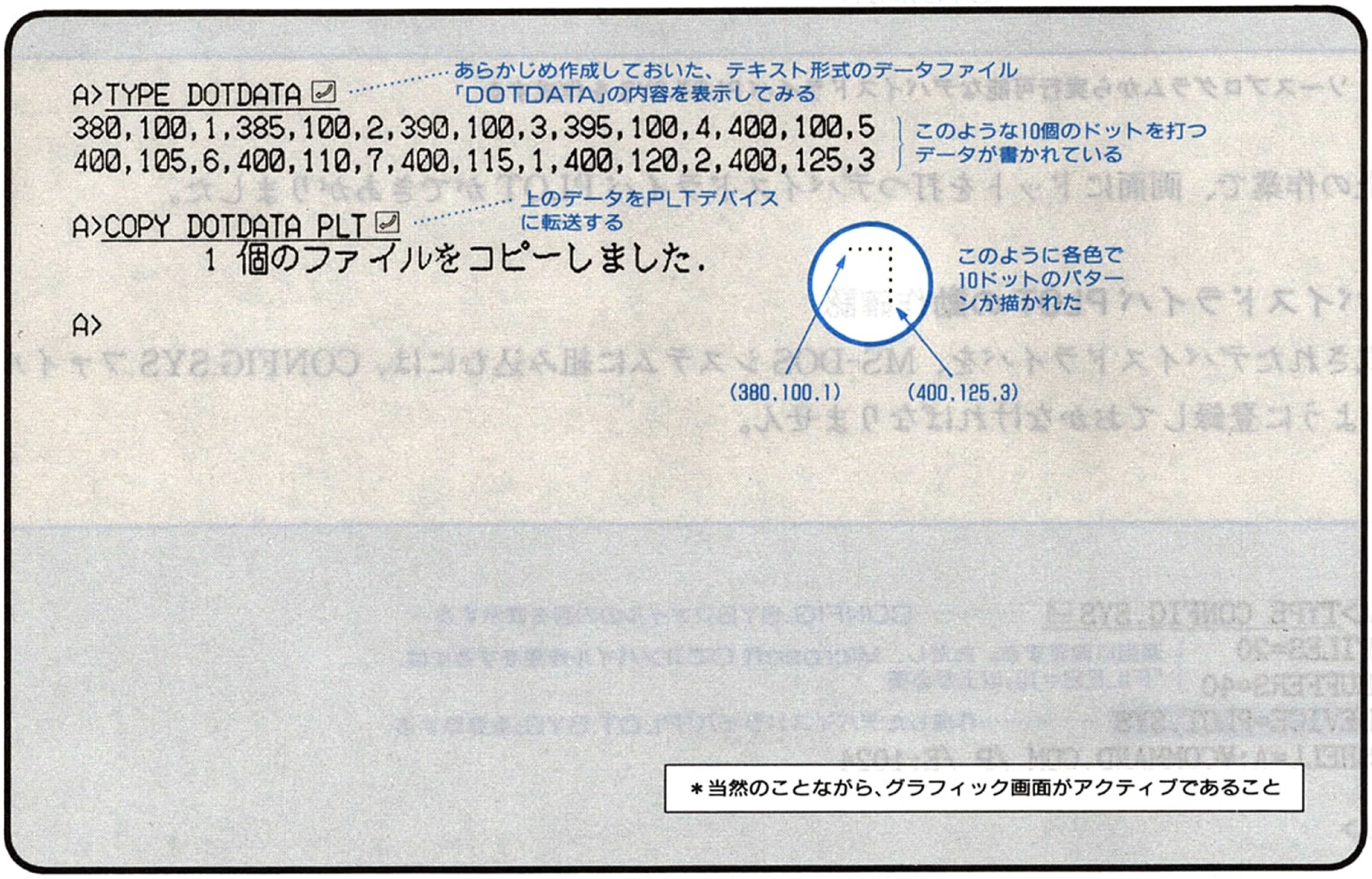
作成したデバイスドライバ「PLOT.SYS」を登録する

図 7.7　作成したデバイスドライバを CONFIG.SYS に登録する

CONFIG.SYSファイルの用意ができたら、デバイスドライバPLOTの動作を確認してみましょう。そのためには次の準備が必要です。

- ●システムディスクのルートディレクトリに、作成したデバイスドライバPLOT.SYSを置く
- ●同じディスクのルートディレクトリに、図7.7のように設定されたCONFIG.SYSファイルを置く
- ●この2つの準備が整えば、このシステムディスクによってMS-DOSを再起動する

以上で、デバイスドライバPLOTを動作させる準備が整いました。MS-DOSが再起動する際に、デバイスドライバPLOTがシステムに組み込まれます。動作テストは、さきほどの図7.4と同じように、CONデバイス(キーボード)からPLTデバイスに座標(x,y)と色のパラメータを送る(コピーする)ことによって行いますが、今回はあらかじめそのデータファイルを作成しておき、そのファイルからPLTデバイスにパラメータを転送してみましょう。図7.8にその実行例を示します。



```
A>TYPE DOTDATA↵
380,100,1,385,100,2,390,100,3,395,100,4,400,100,5
400,105,6,400,110,7,400,115,1,400,120,2,400,125,3

A>COPY DOTDATA PLT↵
        1 個のファイルをコピーしました.

A>
```

あらかじめ作成しておいた、テキスト形式のデータファイル「DOTDATA」の内容を表示してみる

このような10個のドットを打つデータが書かれている

上のデータをPLTデバイスに転送する

このように各色で10ドットのパターンが描かれた

＊当然のことながら、グラフィック画面がアクティブであること

図7.8　デバイスドライバ単体での動作テスト

7.2.2　デバイスドライバの内部解説

前項では、「X座標」「Y座標」「カラーコード」をASCII文字列で与えることにより、指定した座標(X,Y)に、指定した色でドットを打つキャラクタ型デバイスの作成に成功しました。次に、このようなデバイスドライバとはどのようなものか、その内部について簡単に解説しておきましょう。ただし、デバイスドライバのすべてを解説するわけにはいきませんので、詳しくはマニュアルなどを参照していただくとして、ここでは、さきほど作成したデバイスドライバPLOTのような、非常に簡単なキャラクタ型デバイスのドライバを作成するために必要な、最低限の知識を説明するにとどめます。

■ デバイスドライバの実行可能ファイル形式

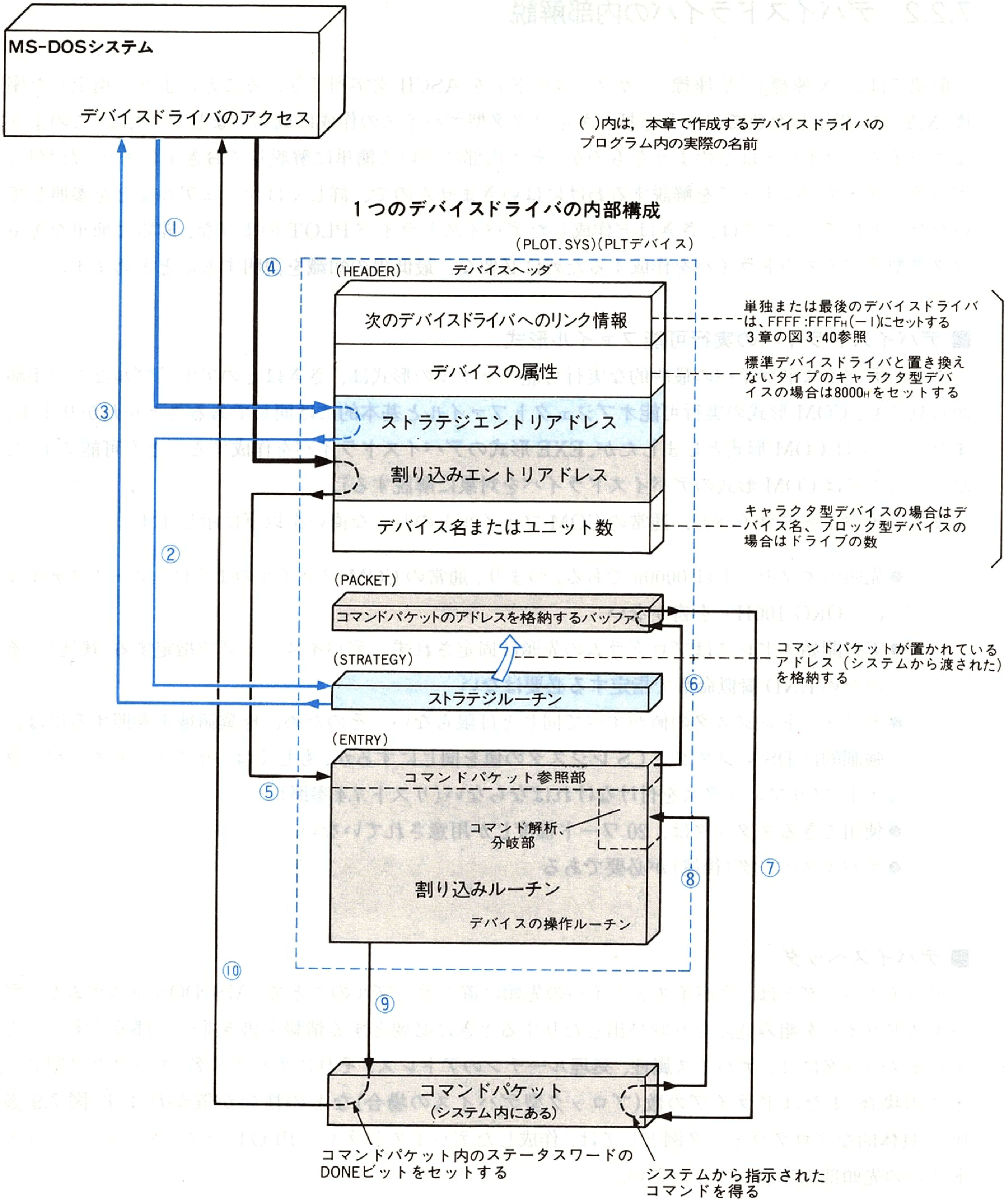
さて、デバイスドライバの最終的な実行可能ファイルの形式は、さきほどのアセンブルなどの手順から見ても、COM形式の実行可能オブジェクトファイルと基本的には同じであることがわかります。また、ここではCOM形式としましたが、EXE形式のデバイスドライバを作成することも可能です(ただし、ここではCOM形式のデバイスドライバを対象に解説する)。

まず、デバイスドライバと、通常のCOMファイルとのおもな違いを以下に示します。

- ●先頭のオフセットは0000Hである。つまり、通常のCOMファイルのように、ソースファイルに「ORG 100H」を書かない
- ●実行開始アドレスはプログラムの先頭に固定されず、デバイスヘッダで指定する(後述)。そのためEND擬似命令で指定する必要はない
- ●セグメントレジスタの値がすべて同じとは限らない。そのため、作業領域を参照するには、強制的にDSレジスタとCSレジスタの値を同じにするか、もしくは、セグメントオーバーライド・プリフィックスを付けなければならない(リスト7.4参照)
- ●使用できるスタックは、20ワード程度しか用意されていない
- ●デバイスヘッダ(後述)が必要である

■ デバイスヘッダ

デバイスヘッダとは、デバイスドライバの先頭に置くテーブルのことで、MS-DOSシステムが、デバイスドライバを組み込んだり呼び出したりするときに必要とする情報を書き並べた部分です。このデバイスヘッダには、デバイス属性、処理ルーチンのアドレス、それにデバイス名(キャラクタ型デバイスの場合)またはドライブの数(ブロック型デバイスの場合)などの情報が置かれます(**図7.9**参照)。具体的なプログラミング例としては、作成したデバイスドライバPLOTのソースファイル(リスト7.1)の先頭部を参照してください。


MS-DOSシステム
デバイスドライバのアクセス
( )内は，本章で作成するデバイスドライバのプログラム内の実際の名前
1つのデバイスドライバの内部構成
(PLOT.SYS)(PLTデバイス)
(HEADER)　デバイスヘッダ
次のデバイスドライバへのリンク情報
デバイスの属性
ストラテジエントリアドレス
割り込みエントリアドレス
デバイス名またはユニット数
単独または最後のデバイスドライバは、FFFF:FFFFH(－1)にセットする
3章の図3.40参照
標準デバイスドライバと置き換えないタイプのキャラクタ型デバイスの場合は8000Hをセットする
キャラクタ型デバイスの場合はデバイス名、ブロック型デバイスの場合はドライブの数
デバイスヘッダ
(PACKET)
コマンドパケットのアドレスを格納するバッファ
(STRATEGY)
ストラテジルーチン
コマンドパケットが置かれているアドレス（システムから渡された）を格納する
(ENTRY)
コマンドパケット参照部
コマンド解析、分岐部
割り込みルーチン
デバイスの操作ルーチン
①
②
③
④
⑤
⑥
⑦
⑧
⑨
⑩
コマンドパケット
(システム内にある)
コマンドパケット内のステータスワードのDONEビットをセットする
システムから指示されたコマンドを得る

① MS-DOSシステムがデバイスドライバへのアクセスを開始する。まず、システムが作成したコマンドパケット（デバイスドライバに要求する処理の内容が書かれている）が置かれているアドレスをデバイスドライバに与え、その内部に格納させるために、デバイスヘッダのストラテジ・エントリアドレスを呼び出す

② ストラテジ・エントリアドレスはストラテジルーチンを指しており、ストラテジルーチンはコマンドパケットの置かれているアドレスをデバイスドライバ内部に格納する処理を行う

③ ②の処理が終わるといったんシステムへ戻る

④ 次にシステムは、コマンドパケットで指示した操作を、デバイスドライバに実行させるための要求を開始する。システムはまず、目的のデバイスを実際に操作する割り込みルーチンを指している、デバイスヘッダの割り込みエントリアドレスを呼び出す

⑤ 割り込みエントリアドレスが指す、割り込みルーチンが実行される

⑥ 割り込みルーチンはまず、コマンドパケットが置かれているアドレスを格納したバッファを参照する

⑦ 得られたコマンドパケットのアドレスにより、コマンドパケットの中身を参照する

⑧ 割り込みルーチンはコマンドパケットの中身を参照してシステムから与えられた処理の内容を知り、割り込みルーチンに戻って該当する処理に分岐し、それを実行する

⑨ システムに要求された処理の実行が終わると、割り込みルーチンはコマンドパケット内のステータスワードのDONEビットをセットする

⑩ デバイスドライバでの処理はすべて終わり、システムに戻る。このあとシステムは、デバイスドライバのアクセスにより得られた結果の処理に移る

図 7.9　デバイスドライバの先頭に置かれるデバイスヘッダの構成と、デバイスドライバの動作の流れ

図 7.9 は、デバイスドライバの内部構成と、デバイスドライバがシステムによってアクセスされたときの動作の流れの概要を示したものです。この図は、あとの解説において随時参照してください。

デバイス属性

デバイス属性とは、キャラクタ型デバイスかブロック型デバイスか、あるいは標準入出力デバイスかなどをMS-DOSに知らせるためのものです。さきほど作成したPLTデバイスのように、標準デバイスと置き換えるタイプではないキャラクタ型デバイスの場合は、デバイス属性を 8000_H にしておきます。

デバイスドライバを呼び出す2つのエントリアドレス

デバイスドライバの処理ルーチンの入口は、**ストラテジ・エントリアドレス**、**割り込みエントリアドレス**の2つが必要で、MS-DOSは、デバイスドライバの先頭のデバイスヘッダを参照してその2つのルーチンのアドレスを知り、それぞれを順に呼び出します。

MS-DOSは、まずストラテジルーチンを呼び出して、これから行う処理を指定するための処理の番号やパラメータが収められている**コマンドパケット**（そのヘッダ部をリクエストヘッダと呼ぶ）のアドレスを渡します。またこのコマンドパケットは、処理コマンド（I／Oリクエストコマンド）の実行が終了してMS-DOSに戻るとき、その結果がセットされることになります。

作成したデバイスドライバPLOTでは、I／Oリクエストコマンド（表7.1参照）のコマンドコード0のINITと、8および9のOUTPUTのみをサポートしています。**図7.10**にOUTPUTを要求する場合のコマンドパケットの内容を示します。

デバイスドライバの2つの処理ルーチンのうち、ストラテジルーチンは、さきに述べたようにMS-DOSからコマンドパケットのアドレスを受け取ります。このストラテジルーチンは、デバイスを操作するような具体的な処理は行わず、MS-DOSから受け取ったコマンドパケットのアドレスをデバイスドライバ内の作業領域に格納するだけで、そのまま戻ります。

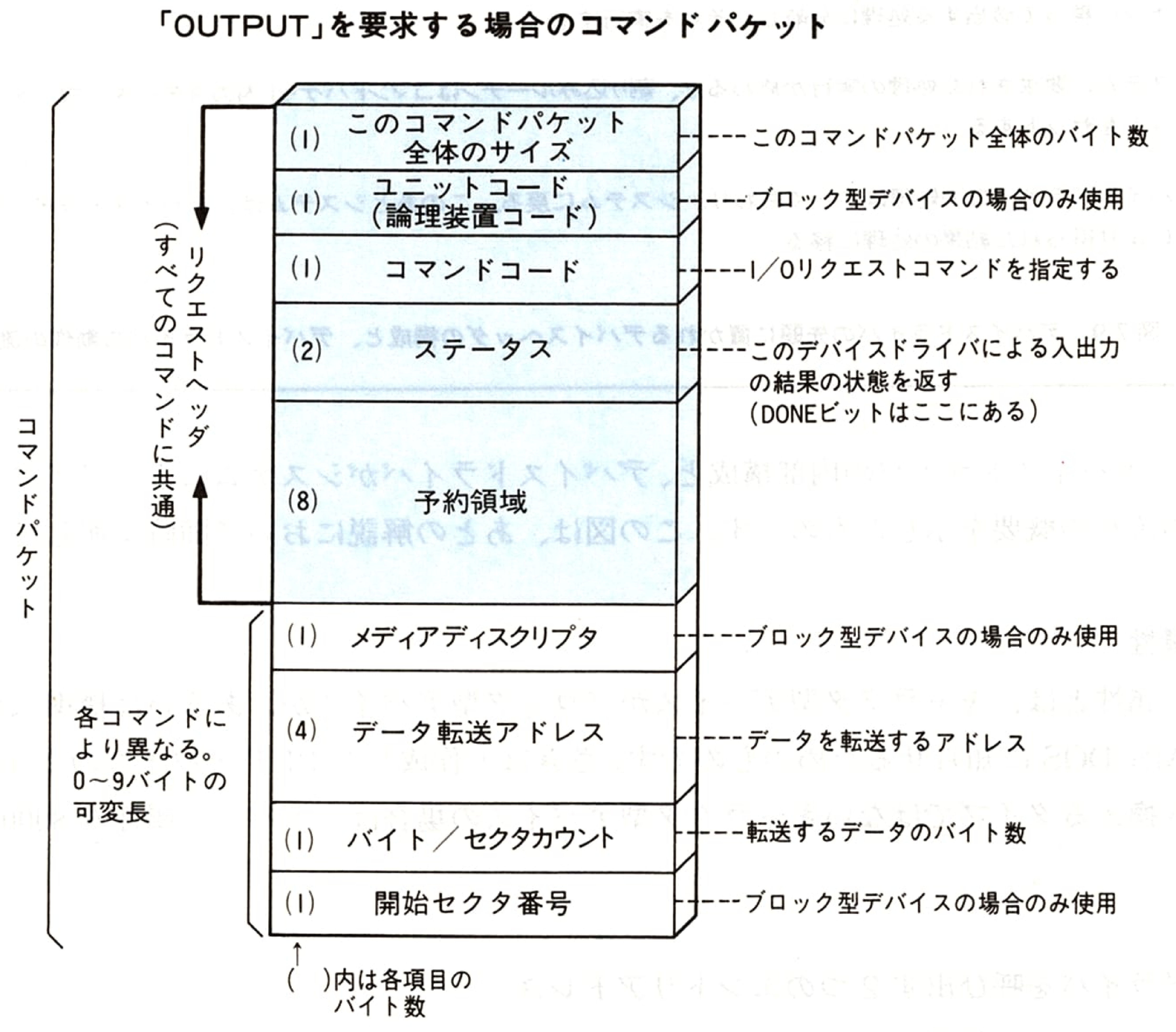

「OUTPUT」を要求する場合のコマンドパケット
(1)　このコマンドパケット全体のサイズ
このコマンドパケット全体のバイト数
(1)　ユニットコード（論理装置コード）
ブロック型デバイスの場合のみ使用
(1)　コマンドコード
I／Oリクエストコマンドを指定する
(2)　ステータス
このデバイスドライバによる入出力の結果の状態を返す（DONEビットはここにある）
(8)　予約領域
リクエストヘッダ（すべてのコマンドに共通）
コマンドパケット
(1)　メディアディスクリプタ
ブロック型デバイスの場合のみ使用
(4)　データ転送アドレス
データを転送するアドレス
(1)　バイト／セクタカウント
転送するデータのバイト数
(1)　開始セクタ番号
ブロック型デバイスの場合のみ使用
各コマンドにより異なる。0～9バイトの可変長
（　）内は各項目のバイト数

図7.10　OUTPUTを要求する場合のコマンドパケットの内容

次に、もう1つの処理ルーチンである割り込みルーチンが呼び出されます。割り込みルーチンは、ストラテジルーチンによって格納されたコマンドパケットのアドレスから、コマンドパケットの内容を参照して、そのコマンドコードで指定されたI／Oリクエストコマンドの処理(入出力などの具体的な処理)を実行します。処理が終わると、コマンドパケット内のステータスワードのDONEビットをセットして戻ります。

ところで、なぜこのように処理ルーチンを2つに分ける必要があるのでしょう。1つのルーチンでも十分処理できそうに思えます。実はそのとおりです。コマンドパケットを渡したあとに、そのまま指定された処理を行えばよいはずであり、現在のMS-DOSでは、2つに分ける必要はまったくありません。それどころか処理が繁雑になり、場合によっては処理速度の低下を招きます。

それなのにわざわざ2つに分けているのは、将来のマルチタスクへの対応を考えていたからです。MS-DOSはシングルタスクですが、マルチタスクをサポートするためには、このように処理ルーチンを2つに分けることが必要になります。マルチタスクの環境では、あるタスクの入出力の処理中に、別のタスクの入出力要求が起こるかもしれません。こうなった場合、ハードウェアレベルで同時に2つの入出力を行うことはできないため、ストラテジルーチンで、それらの要求をキュー(要求待ちバッファ)にためておき、割り込みルーチンでそれらを順にキューから取り出して処理するという手順が必要になるのです。

MS-DOSは、将来のマルチタスク化を考慮して設計されていたわけですが、結局そのままの形でマルチタスクを実現することはできませんでした。しかし、マルチタスクの思想は、次の新しいOSであるOS/2に引き継がれ、そこで実現されています。

I／Oリクエストコマンドの実行

さて、デバイスドライバ内のこの2つの処理ルーチンは、MS-DOSによってFARコールされます。ですから、ソースファイル上のこの2つのルーチンは、FARタイプのPROCにしておきます(作成したデバイスドライバのリスト7.1参照)。

まず最初のストラテジルーチンは、コマンドパケットのアドレスを作業領域に格納してすぐ戻ります。次の割り込みルーチンは、そのアドレスからコマンドパケットの内容を参照し、コマンドコードで指定されたI／Oリクエストコマンドによる処理を行います。コマンドコードの種類は、MS-DOSのバージョン2.*x*で13種類、バージョン3.1で17種類、バージョン3.3では20種類ありますが、ここで作成するプログラムは、バージョン2.*x*に対応して書かれています(**表7.1**参照)。

コマンドコード0のINITはデバイスを初期化するもので、このコマンドは、MS-DOSの起動時に、デバイスドライバがシステムに組み込まれる際に1度だけ呼ばれます。このコマンドが指定されると、デバイスを初期化したあと、デバイスドライバの終了アドレスを返します。これはデバイスドライバが常駐するための大きさを指定するものです。

ところで、このことを利用してメモリを節約することができます。初期化ルーチンをデバイスドラ

コマンドコード	デバイス・ドライバの形式	コマンド	機　能
0	B／C	INIT	デバイスドライバの初期化を行う
1	B	MEDIA CHECK	ディスクが交換されたかどうかを調べる
2	B	BUILD BPB	現在アクセスされているディスクに対応するBPBを作成する
3	B／C	IOCTL INPUT	デバイスドライバ自身からデータを入力する
4	B／C	INPUT	デバイスからデータを入力する（READ）
5	C	NON-DESTRUCTIVE INPUT NO WAIT	入力バッファの先頭の1バイトの内容を調べる
6	C	INPUT STATUS	入力バッファの内容が空かどうか調べる
7	C	INPUT FLUSH	入力バッファを空にする
8	B／C	OUTPUT	デバイスへデータを出力する（WRITE）
9	B／C	OUTPUT WITH VERIFY	デバイスへデータを出力するだけではなく、正しく出力できたかどうかの検査も行う
10	C	OUTPUT STATUS	出力バッファ内にデータが残っているかどうかを調べる
11	C	OUTPUT FLUSH	出力バッファの内容を空にする
12	B／C	IOCTL OUTPUT	デバイスドライバ自身へデータを渡す
13	B／C	DEVICE OPEN	デバイスのオープン。ただし、デバイスドライバがOPEN／CLOSE／RMの機能を持つもののみ有効
14	B／C	DEVICE CLOSE	デバイスのクローズ、条件は13と同じ
15	B	REMOVABLE MEDIA	メディアが交換可能なデバイスかどうかを調べる。 条件は13と同じ
16	C	OUTPUT UNTIL BUSY	デバイスがBUSY状態になるまで出力を続ける
19		GENERIC IOCTL	デバイス属性のビット6が1のデバイスのみ有効
23		GET DRIVE MAP	デバイス属性のビット6が1のデバイスのみ有効
24		SET DRIVE MAP	デバイス属性のビット6が1のデバイスのみ有効

B………ブロック型デバイスに対する機能
C………キャラクタ型デバイスに対する機能
B／C… 共通の機能

□……本章で作成するデバイスドライバでサポートするコマンド、ただしコマンドコード9のベリファイの機能はなし
■……MS-DOSバージョン3.*x*で拡張された機能

表7.1　I／O リクエストコマンドの種類

イバの最後部に置き、その初期化ルーチンの前に終了アドレスを置けば、初期化ルーチンの占めるメモリ領域は初期化時のみに使われ、そのあとほかの目的のために解放することができるのです。

さて次は、作成したデバイスドライバの、コマンドコード8および9のOUTPUTを処理する場合について考えてみましょう。このコマンドコード9は、実は書き込みと同時にそれが正しく書き込まれたかどうかを検査（ベリファイ）するためのコマンドなのですが、ここでは省略しています。

デバイスに書き込むためのデータを得るには、コマンドパケットの14バイト目に格納されている転送データの先頭アドレスと、コマンドパケットの18バイト目に格納されている転送するバイト数の2つのアドレスを参照します。このようにしてデータを得て処理を行い、すべてが終了したら、コマンドパケットの4バイト目のステータスワードのDONEビットをセットして戻ります。もしすべての

データを処理しきれなかった場合には、処理できたバイト数を、18バイト目のバイトカウントにセットしてから戻ります。

なお、すべての割り込みルーチンの実行に際して、各レジスタの内容は保護されませんので、必要なレジスタには保存処理が必要になります。注意してください。

以上はキャラクタ型デバイスについての説明ですが、ブロック型デバイスについても基本的には同じです。ただし、ブロック型デバイスの場合は、2章のファイルシステムの、とくにディスク領域の配置についての知識が必要になります。

7.2.3　デバイスドライバのアクセス法

前項で、デバイスドライバの先頭部に置かれている、コマンドやパラメータを受け渡しするデバイスヘッダについて解説してきましたが、ユーザープログラムからデバイスドライバをアクセスするには、具体的にどのようにすればよいのでしょう。

たとえば、デバイスに書き込みを行う場合について考えてみましょう。すでにおわかりかもしれませんが、これはディスク上のファイルに書き込みを行う場合と同じで、そのデバイスをオープンしてから書き込み、最後にクローズして終わります。MS-DOSでは、デバイスもファイルとまったく同等に取り扱われるため、デバイスに対しても通常のディスクファイルに対する入出力と同じ操作で読み書きすることができるのです。

ここで作成したデバイスドライバPLOTの場合はPLTというデバイスをオープンし、それに対して書き込みを行えば、グラフィック画面にドットが打たれます。オープンや書き込みの方法などは、2章や4章で解説したとおりです。

7.2.4　デバイスドライバPLOTを利用するユーザープログラムの作成

さきに作成したデバイスドライバPLOTを利用して、フリーハンドで線を描く、お絵かきプログラムを作ってみましょう(図7.2参照)。

このプログラムの仕様は、まずプログラムを起動するとマウスのカーソル「↖」が表示され、左ボタンを押しながらマウスを移動すると、それに従ったマウスカーソルの先端の軌跡が、白いドット(色は、プログラム内の指定による)で描かれます。ボタンを押さずにマウスを移動した場合は、軌跡は描かれません。このプログラムはマウスの右ボタンを押すことによって終了し、MS-DOSのプロンプトに戻ります。

このプログラムはC言語で記述してありますので(アセンブラ版も示す)、マウスドライバのようなMS-DOSの内部割り込みによりアクセスするデバイスドライバを、C言語のユーザープログラムから

呼び出す場合のよいサンプルにもなるでしょう。

このプログラム全体は、次の5つのファイルから構成されています。

trace.c	メインモジュール
graphic.h	描画関数用ヘッダ
graphic.c	グラフィックドライバ呼び出し関数モジュール
mouse.c	マウスドライバ呼び出し関数モジュール
mouse.h	マウス用構造体定義ヘッダ

メインモジュールのファイル名がtraceですので、このプログラムの名前を「TRACE」ということにして、**リスト7.5**、**リスト7.6**、**リスト7.7**、**リスト7.8**、**リスト7.9**にそれぞれのソースファイルを示します。

```
/*
 * trace.c        trace mouse program
 */

#include <stdio.h>
#include "mouse.h"
#include "graphic.h"

void main(void)
{
        unsigned short x, y;
        MOUSE mouse;

        if (gropen() < 0) {
                fputs("Can't open graphics¥n", stderr);
                exit(1);
        }
        if (mouopen() < 0) {
                fputs("Can't open mouse¥n", stderr);
                exit(1);
        }
        mouon();
        moustat(&mouse);
        x = mouse.x;
        y = mouse.y;
        if (mouse.sw & Mask(MouLeftButton)) {
                mouoff();
                grplot(x, y, 7);
                mouon();
        }
```

void型がサポートされていない処理系では、voidをintに置き換えてください（#define void intの１行を追加する）

オープンできなかった場合、エラーメッセージを表示してこのプログラムを終了する（CONFIG.SYSファイルにPLTデバイスが登録されていなかった場合など）

終了コード「1」を返して終了する

マウスカーソルを表示する

— リスト 7.5 — (次ページに続く)

```
        for (;;) {
                moustat(&mouse);
                if (mouse.sw & Mask(MouRightButton)) {
                        break;
                } else if (mouse.sw & (Mask(MouLeftButton))) {
                        if (x != mouse.x || y != mouse.y) {
                                x = mouse.x;
                                y = mouse.y;
                                mouoff();
                                grplot(x, y, 7);
                                mouon();
                        }
                }
        }
        mouoff();
        mouclose();
        grclose();
        exit(0);
}

/* end of trace.c */
```

moustat(&mouse); ……マウスの状態を得る

if (mouse.sw & Mask(MouRightButton)) { ……右のスイッチが押されていたらプログラムを終了する

} else if (mouse.sw & (Mask(MouLeftButton))) { …左のスイッチが押されていて、

if (x != mouse.x || y != mouse.y) { ……マウスが移動していたら、

mouoff(); ……マウスカーソルを消してドットを打つ処理へ

grplot(x, y, 7); 「白」を設定

mouoff(); ……マウスカーソルを消す

grclose(); ……PLTデバイスをクローズする

マウスカーソルのON／OFFについて

グラフィック画面を操作する際には、表示されているマウスカーソルをかならず消さなければならない

リスト 7.5　メインモジュール trace.c

```
/*
 * graphic.h
 */

int gropen(void);
void grclose(void);
void grplot( unsigned short x,
             unsigned short y,
             unsigned short c );
```

リスト 7.6　描画関数用ヘッダ graphic.h

```
/*
 * graphic.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <fcntl.h>
#include "graphic.h"

static char devname[] = "PLT";
static int fd;

int gropen(void)
{
        fd = open(devname, O_WRONLY);
        if (fd < 0)
                return -1;
        return 0;
}

void grclose(void)
{
        close(fd);
}

void grplot( unsigned short x,
             unsigned short y,
             unsigned short c )
{
        char buf[20];

        sprintf(buf, "%4u %4u %2u¥n", x, y, c);
        write(fd, buf, strlen(buf));
}

/* end of graphic.c */
```

……PLTデバイスを書き込みモードでオープンする

PLTデバイスへ座標値とカラーコードを指定した文字列を書き込む。これによってドットが打たれる

リスト 7.7　グラフィックドライバ呼び出し関数モジュール graphic.c

```
/*
 * mouse.c        mouse control functions
 */

#ifdef DEBUG
#include <stdio.h>
#endif
#include <dos.h>
#include "mouse.h"
```

マウスドライバを呼び出し、現在の座標やボタンの状態などを得る関数

— リスト 7.8 —（次ページに続く）

```
static union REGS reg;  ……………CPUのレジスタを表す構造体

int mouopen(void)
{
        reg.x.ax = 0x00;
        int86(0x33, &reg, &reg);
        if (reg.x.ax == 00)
                return -1;
        return 0;
}

void mouclose(void)
{
        mouoff();
}

void mouon(void)
{
        reg.x.ax = 0x01; ……機能コード1(マウスカーソルの表示)をAXレジスタにセットする   マウスカーソル
        int86(0x33, &reg, &reg); ……………割り込み INT 33Hを実行する                  の表示
}

void mouoff(void)
{
        reg.x.ax = 0x02; ……機能コード2(マウスカーソルの消去)をAXレジスタにセットする   マウスカーソル
        int86(0x33, &reg, &reg); ………… INT 33Hを実行する                            の消去
}

void moustat(MOUSE *mptr)
{
                           …… 機能コード3(マウスカーソルの状態を得る)を
        reg.x.ax = 0x03;       AXレジスタにセットする
        int86(0x33, &reg, &reg); ………… INT 33Hを実行する
        mptr->x = reg.x.cx; ……………CXレジスタにX座標値が返される                      マウスカーソル
        mptr->y = reg.x.dx; ……………DXレジスタにY座標値が返される                      の状態を得る
        mptr->sw = (reg.x.bx&0xff); …………BXレジスタにマウスの状態が返される
#ifdef DEBUG
        fprintf(stderr, "x=%3d, y=%3d, sw=%2X¥n", mptr->x, mptr->y, mptr->sw);
#endif
}                                            0……押されていない
                                             1……左が押されている
                          マウスボタンの状態は、 2……右が押されている
/* end of mouse.c */                         3……両方が押されている
```

リスト 7.8　マウスドライバ呼び出し関数モジュール mouse.c

```
/*
 * mouse.h      definition of MS mouse interface
 */

typedef struct {
        unsigned short x;
        unsigned short y;
        unsigned short sw;
} MOUSE;

#define MouLeftButton 0
#define MouRightButton 1
#define MouCenterButton 2
#define Mask(s) (1 << (s))

int mouopen(void);
void mouclose(void);
void mouon(void);
void mouoff(void);
void moustat(MOUSE *mptr);
```

マウスの状態を格納する構造体の定義

マウスカーソルの指すＸＹ座標と、マウスの状態を格納する

マウスボタンのマスク

リスト 7.9　マウス用構造体定義ヘッダファイル mouse.h

エディタでこれらのソースファイルを作成します。コンパイルおよびリンクの手順は、6章のchmodプログラムの実行例と同じです。図 7.11 にその実行例を示しますが、詳しくは6章を参照してください。

さて以上の作業で、実行可能な trace プログラム TRACE.EXE ができあがりました。このプログラムを実行するには、次の2つのデバイスドライバを用意します。

- マイクロソフトマウス用に提供されているマウスドライバ MOUSE.SYS*
- 本章の 7.2.1 で作成したドットを打つデバイスドライバ PLOT.SYS

また、図 7.7 に「DEVICE＝MOUSE.SYS」の1行を追加した CONFIG.SYS ファイルの用意ができたら、お絵かきプログラム trace を実行するために次の準備を行います。

- システムディスクのルートディレクトリに、マウスドライバ MOUSE.SYS を置く
- システムディスクのルートディレクトリに、デバイスドライバ PLOT.SYS を置く
- 同じディスクのルートディレクトリに、MOUSE.SYS を登録した CONFIG.SYS ファイルを置く

---

* 各機種に付属のマウスドライバを使用する場合は、mouse.c に、若干の修正を加える必要がある。

Microsoft Cによる例(6章の 図6.14参照)。

```
A>SET ⏎ ……………環境変数をチェックする
COMSPEC=A:¥COMMAND.COM
PATH=A:¥BIN;A:¥MSC5¥RBIN;A:¥MSC5¥BIN
INCLUDE=A:¥MSC5¥INCLUDE ……………これがセットされていると、Microsoft Cはインクルードファイルを
                                このディレクトリから探してくれる
LIB=A:¥MSC5¥LIB ……
TMP=A:¥TMP        これがセットされているとLINKバージョン3.xはこのディレクトリからライブラリを捜してくれる。
                  たとえば、「B：¥LIB;C：¥USR¥LIB」のようにサーチパスを並べることもできる

A>CL TRACE.C MOUSE.C GRAPHIC.C ⏎ ……………この2つのソースファイルに対してコンパイル&リンクを行う
Microsoft (R) C Optimizing Compiler Version 5.10
Copyright (c) Microsoft Corp 1984, 1985, 1986, 1987, 1988. All rights reserved.

TRACE.C   ⎫
MOUSE.C   ⎬ この3つのソースファイルに対する処理が行われている
GRAPHIC.C ⎭

Microsoft (R) Overlay Linker  Version 3.65 ……コンパイルが終了すると自動的にリンカが起動される
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Object Modules [.OBJ]: TRACE.OBJ +
Object Modules [.OBJ]: MOUSE.OBJ +
Object Modules [.OBJ]: GRAPHIC.OBJ
Run File [TRACE.EXE]: TRACE.EXE /NOI
List File [NUL.MAP]: NUL         ↑
Libraries [.LIB]:                └──大文字／小文字の区別をするためのオプション
          ↑
          └──ライブラリを指定していないことに注目
A>           (OBJファイル中にその名前は埋め込まれているので)
```

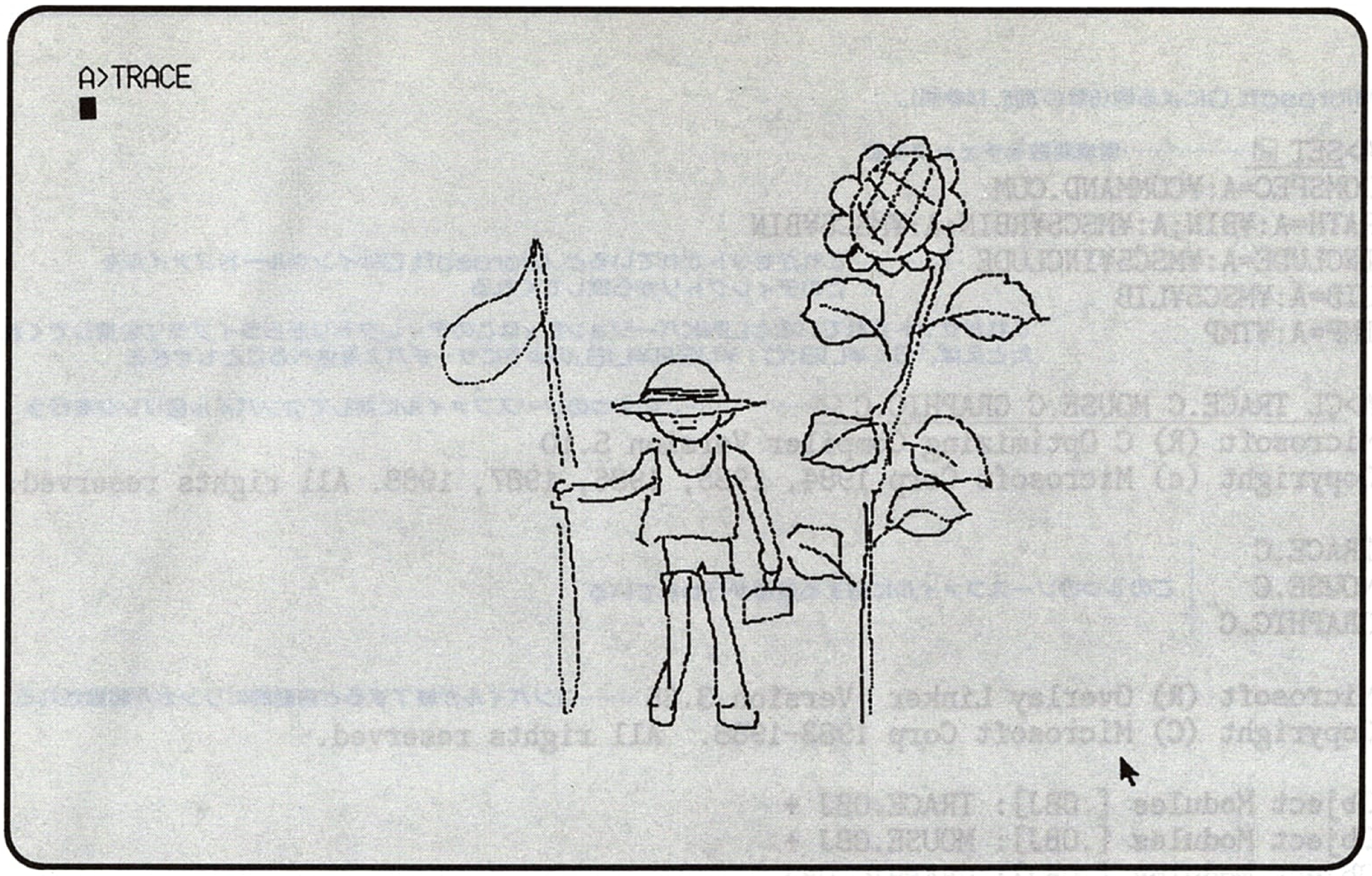
図7.11　ソースファイルのコンパイル&リンク作業の実行例

- ●これらの準備が整えば、このシステムディスクによってMS-DOSを再起動する
- ●グラフィック画面をアクティブにする(前出のGRONプログラムや、その他の方法で)

このようにしてMS-DOSを起動すれば、お絵かきプログラムTRACE.EXEは、どこのディレクトリにあってもかまいません。

```
A>TRACE ⏎
```

と入力して、このプログラムを起動すると、マウスカーソルが表示されますので、左ボタンを押しながらマウスを移動すればその軌跡が描かれるでしょう(図7.12参照)。



```
A>TRACE
█
```

図 7.12　trace プログラムを実行して描いた絵

7.3　アセンブラによるお絵かきプログラムの作成

前項では、trace プログラムをC言語で作成しましたが、ここでは参考までに、そのプログラムをアセンブラで書き直してみましょう。プログラムの仕様などは、C言語で作成したものとまったく同じであり、それをただアセンブラでプログラミングするだけです。

アセンブラの場合でも、デバイスドライバをアクセスする手順は同じです。システムコールを呼び出して、PLT デバイスをオープンし、同じくシステムコールでそのファイルに書き込みを行えば、デバイスに対して書き込みを行うことができます。

このアセンブラ版お絵かきプログラムの、ソースファイル TRACEA.ASM を**リスト 7.10** に示します。ソースファイルはこれ1本のみであり、これから作成される実行可能オブジェクトファイル TRASEA.COM（**図 7.13** 参照）と、7.2 節で作成したデバイスドライバ PLOT.SYS があれば、このプログラムは動作します。

```
;; TRACEA.ASM      Trace Program
;;                  Asm Version
;;
;;      09H        Display String
;;      3DH        Open a File
;;      3EH        Close a File Handle
;;      40H        Write to File/Device
;;      4CH        Terminate a Process

CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE
        ORG 100H
START:
        MOV     AH,3DH      ファイルオープンのファンクションリクエスト
        LEA     DX,GRAPH     AH=3DH                                     PLT(デバイス)ファイルをオープンする
        MOV     AL,1         DS:DX=パス名のアドレス
        INT     21H          AL=アクセスモード
        JC      OERROR   ……………キャリーフラグが「1」ならオープンエラー
        MOV     HANDLE,AX
        MOV     AX,1        マウスカーソルを表示する(マウスドライバをコールする)
        INT     33H
LOP:
        MOV     AX,3        マウスの状態を得る
        INT     33H
        TEST    BX,2        右ボタンが押されていたらこのプログラムを終了する
        JNZ     BREAK
        TEST    BX,1
        JZ      LOP
        CMP     CX,X        左ボタンが押されていて、マウスが移動していた場合にはドットを打つ
        JNE     PLOT        (マウスカーソルが同じ位置でチラチラしないように)
        CMP     DX,Y
        JE      LOP
PLOT:
        MOV     X,CX              得られた座標値(X,Y)と指定するカラーコードをASCII文字列に変換して、
        MOV     Y,DX              PLTデバイスへ書き込む処理
        LEA     SI,COLOR
        LEA     DI,BUF+4*3-1
        MOV     BX,3
CONV:
        MOV     BYTE PTR [DI],' '
        DEC     DI                バイナリ数値→ASCII文字列の変換。
        CALL    ITOA              X座標、Y座標、カラーコードのそれぞれについて行う
        DEC     BX
        JNE     CONV
        MOV     AX,2        マウスカーソルを消す
        INT     33H
        MOV     AH,40H      ファイルへの書き込みのファンクションリクエスト
        LEA     DX,BUF       AH=40H                                     PLTデバイスに座標値、カラー
        MOV     CX,4*3       DS:DX=パス名のアドレス                      コードを示す文字列を書き込む。
        MOV     BX,HANDLE    CX=書き込みバイト数                         これによってドットが打たれる
        INT     21H          BX=ファイルハンドル
```

— リスト 7.10 — (次ページに続く)

```
          JC      WERROR ……………キャリーフラグが1ならエラー処理へ
          MOV     AX,1   ┐
          INT     33H    ┘マウスカーソルを表示する
          JMP     SHORT LOP ……………これまでの処理の繰り返し
OERROR:                     ┐
          LEA     DX,OERRMES│ファイルオープンエラーの処理
          JMP     SHORT MESOUT┘
WERROR:                     ┐
          LEA     DX,WERRMES┘ファイル書き込みエラーの処理
MESOUT:
          MOV     AH,9   文字列出力のファンクションリクエスト        ┐
          INT     21H    {AH=09H                                      │エラーメッセージを出力する
                         {DS:DX=文字列の先頭アドレス                  ┘
          JMP     SHORT RETURN
BREAK:
          MOV     AX,2   ┐
          INT     33H    ┘マウスカーソルを消去する
          MOV     AH,3EH      ファイルクローズのファンクションリクエスト ┐
          MOV     BX,HANDLE   {AH=3EH                                    │PLTデバイスをクローズする
          INT     21H         {BX=ファイルハンドル                       ┘
RETURN:
          MOV     AX,4C00H ┐
          INT     21H      ┘プロセスの終了
ITOA      PROC          ┐                                  ┐
          MOV     AX,[SI]│                                  │
          DEC     SI    │                                  │
          DEC     SI    │                                  │
          MOV     CX,3  │                                  │
          MOV     DL,10 │SIレジスタの指す位置のバイナリ数値を│
TOASC:                  │3桁の10進文字列に変換する。         │バイナリ数値を10進文字列
          DIV     DL    │結果はDIレジスタの指すアドレスから順│に変換する関数
          ADD     AH,'0'│に格納する                          │
          MOV     [DI],AH│                                 │
          DEC     DI    │                                  │
          XOR     AH,AH │                                  │
          LOOP    TOASC │                                  │
          RET           ┘                                  ┘
ITOA      ENDP
X         DW      -1 ……………X座標        ┐
Y         DW      -1 ……………Y座標        │バイナリ数値を格納する
COLOR     DW      7 ……………カラーコード  ┘
BUF       DB      '000 000 000 ' ……………10進文字列を格納する
HANDLE    DW      0 ……………ファイルハンドルを格納する
GRAPH     DB      "PLT",0 ……………(デバイス)ファイル名
OERRMES   DB      0DH,0AH,"OPEN ERROR",0DH,0AH,'$'  ┐
WERRMES   DB      0DH,0AH,"WRITE ERROR",0DH,0AH,'$' ┘エラーメッセージ
CODE      ENDS
          END     START
```

リスト7.10　お絵かきプログラムのアセンブラ版ソースプログラム TRACEA.ASM

```
A>MASM TRACEA,,TRACEA;⏎ ………アセンブリ・ソースファイル「TRACEA.ASM」をアセンブルし、
                            「TRACEA.OBJ」および「TRACEA.LST」を生成する
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


 47204 + 257080 Bytes symbol space free

     0 Warning Errors
     0 Severe  Errors

A>LINK TRACEA;⏎ ……………アセンブルにより生成されたTRACEA.OBJに対してリンカを実行する

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

LINK : warning L4021: no stack segment ……………COM形式なのでこのエラーは気にしなくてよい

A>EXE2BIN TRACEA.EXE TRACEA.COM⏎ ……………「TRACEA.EXE」を「TRACEA.COM」に変換する

A>                 実行可能なプログラム「TRACEA.COM」が完成した
```

図 7.13　TRACE プログラムのアセンブルおよびリンクの実行例

以上の作業で、実行可能なお絵かきプログラム TRACEA.COM ができあがりました。これを実行するための準備は、C 言語で作成した trace プログラムの場合とまったく同じです。ではプログラムを実行してみましょう。今回は

```
A>TRACEA⏎
```

と入力することにより、さきほどと同じ動作が行われます。

7.4　デバイスドライバを使わないお絵かきプログラムの作成

ここでは、7.2 や 7.3 で作成したものと同じ機能を持つお絵かきプログラムを、デバイスドライバ PLOT を利用せずに、ユーザープログラムの中だけで処理する形式（図 7.3 参照）で作成してみましょう。このプログラムは、さきのプログラムではデバイスドライバを呼んで処理していた部分を、ユーザープログラム内にアセンブラルーチンで用意し、それを C 言語のメインモジュールから呼び出す形で実現します。このプログラム全体は、次の 5 つのファイルから構成されます。

mouse.h、mouse.c、trace.c …この3つは7.2節で作成したものと同じ
GRAPHIC2.ASM ………………デバイスドライバPLOTの代わりとなる、ドットを打つ関数

最初の3つは、7.2節のものと同じですので省略し、**リスト7.11**に「GRAPHIC2.ASM」のソースプログラムを示します。

▨の部分以外はMicrosoft Cコンパイラが出力したもの。出力されたリストにこの枠内のプログラムを書き込んで完全な関数とする。つまり、関数の入口、出口の「枠」やセグメント名などをMicrosoft Cに作成してもらい、関数の中身だけを自分で書く、という手法である(後述)

```
;       Static Name Aliases
;
        TITLE   graphic2.c
        NAME    graphic2

        .8087
_TEXT   SEGMENT  WORD PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
$$SYMBOLS       SEGMENT  BYTE PUBLIC 'DEBSYM'
$$SYMBOLS       ENDS
$$TYPES SEGMENT  BYTE PUBLIC 'DEBTYP'
$$TYPES ENDS
DGROUP  GROUP   CONST, _BSS, _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP
EXTRN   __acrtused:ABS
EXTRN   __chkstk:NEAR
_TEXT      SEGMENT
        ASSUME  CS: _TEXT
; Line 9
; Line 8
        PUBLIC  _gropen
_gropen PROC NEAR
        push    bp
        mov     bp,sp
        xor     ax,ax
        call    __chkstk
; Line 9
        sub     ax,ax
        jmp     SHORT $EX106
; Line 10
$EX106:
        mov     sp,bp
        pop     bp
        ret

_gropen ENDP
```

ここにはもともと「EXTRN _dummy : NEAR」があったが、これを削除する

— リスト 7.11 — (次ページ以下に続く)

```
; Line 13
        PUBLIC  _grclose
_grclose        PROC NEAR
        push    bp
        mov     bp,sp
        xor     ax,ax
        call    __chkstk
```

フレームポインタのセットアップおよびスタックチェック

```
; Line 14
        mov     sp,bp
        pop     bp
        ret

_grclose        ENDP
; Line 19
        PUBLIC  _grplot
_grplot PROC NEAR
        push    bp
        mov     bp,sp
        xor     ax,ax
        call    __chkstk
;       x = 4
;       y = 6
;       c = 8
; Line 20
```

ここにはもともと「call _dummy」があったが、それを削除してアセンブラルーチンと置き換える

```
INCLUDE CONFIG.INC
;; GRAPH.ASM    Graphic Plane Plot
;;              Procedures

IF MACHINE EQ PC9801

HSIZE   EQU     640
VSIZE   EQU     400
NPLANE  EQU     3

BGVRAM  SEGMENT AT 0A800H
BGVRAM  ENDS
RGVRAM  SEGMENT AT 0B000H
RGVRAM  ENDS
GGVRAM  SEGMENT AT 0B800H
GGVRAM  ENDS

ENDIF ; PC9801

IF MACHINE EQ FMR60

HSIZE   EQU     1120
VSIZE   EQU     750
NPLANE  EQU     3

LIMIT   EQU     400
```

```
GVRAM0  SEGMENT AT 0C000H
GVRAM0  ENDS
IF ((LIMIT * (HSIZE / 8)) AND 0FH) NE 0
ERROR
ENDIF
GVRAM1  SEGMENT AT 0C000H + ((LIMIT * (HSIZE / 8)) SHR 4)
GVRAM1  ENDS

ENDIF ; FMR60

PX      EQU     4
PY      EQU     6
COL     EQU     8

        PUSH    SI
        PUSH    DI
        PUSH    DS
```

Microsoft CではセグメントレジスタおよびレジスタとしてSIレジスタ、DIレジスタを保存しなければならない

```
        MOV     AX,[BP].PX
        MOV     BX,[BP].PY
```

引数を取り出す。(すでにBPをフレームポインタとして使用できるように設定されている)

```
        CALL    COMPADR
```

V-RAMアドレスおよびビットアドレスを計算する

```
        MOV     BX,[BP].COL
        MOV     CX,NPLANE
IF MACHINE EQ PC9801
BITSET: MOV     DS,CS:[SI]
        INC     SI
        INC     SI
ENDIF ; PC9801
IF MACHINE EQ FMR60
        MOV     AX,1B
BITSET: MOV     SI,DX
        MOV     DX,0402H
        OUT     DX,AL
        SHL     AL,1
        XCHG    AL,AH
        MOV     DX,0404H
        OUT     DX,AL
        XCHG    AL,AH
        INC     AH
        MOV     DX,SI
ENDIF ; FMR60
        SHR     BL,1
        JNC     CLEAR
        OR      [DI],DL
```

RGB各プレーンごとに、カラーコードに従ってビットをON／OFFする

```
        LOOP    BITSET
        JMP     RETURN
CLEAR:
        AND     [DI],DH
        LOOP    BITSET
RETURN:
        MOV     DS,[BP-6]
        MOV     DI,[BP-4]
        MOV     SI,[BP-2]
```

レジスタを復帰する。なお、この関数は値を返さないが、返す場合はAXレジスタに格納する(int型など)

```
          MOV     SP,BP
          POP     BP
          RET

COMPADR PROC      NEAR
IF MACHINE EQ PC9801
          MOV     SI,OFFSET SEGTBL
ENDIF ; PC9801
IF MACHINE EQ FMR60
          MOV     SI,GVRAM0
          CMP     BX,LIMIT
          JB      CPADR
          SUB     BX,LIMIT
          MOV     SI,GVRAM1
CPADR:    MOV     DS,SI
ENDIF ; FMR60
          MOV     CX,AX
          MOV     AX,HSIZE / 8
          MUL     BX
          MOV     DI,AX
          MOV     AX,CX
          SHR     AX,1
          SHR     AX,1
          SHR     AX,1
          ADD     DI,AX
          AND     CL,111B
          MOV     DL,10000000B
          SHR     DL,CL
          MOV     DH,DL
          NOT     DH
          RET

IF MACHINE EQ PC9801
SEGTBL    DW      RGVRAM
          DW      GGVRAM
          DW      BGVRAM
ENDIF ; PC9801

COMPADR ENDP
; Line 21
          mov     sp,bp
          pop     bp
          ret

_grplot ENDP
          nop
_TEXT     ENDS
END
```

……………フレームポインタの復帰

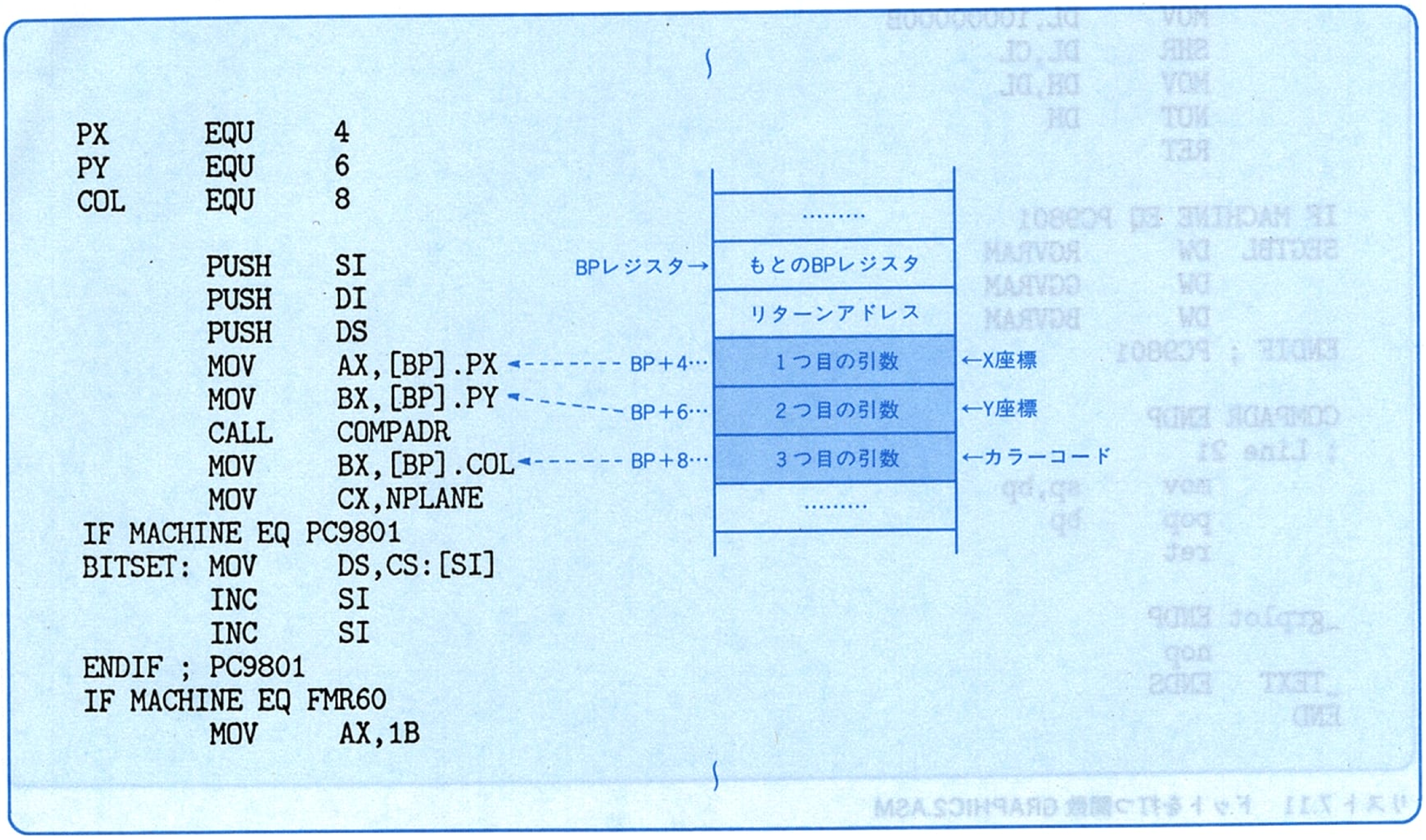
「GRAPH.ASM」のものと同じ。ただし、セグメントの宣言は取り除く

リスト 7.11　ドットを打つ関数 GRAPHIC2.ASM

アセンブラで書いた関数をC言語から呼び出すためには、アセンブラのソースファイルを書く際に、いろいろな約束事があります。それらのおもなものを以下に挙げておきましょう。

- セグメント名をC言語のものと一致させる。ただし、ラージモデルではその必要のない場合もある
- レジスタの値を保存する。とくにセグメントレジスタ、BPレジスタは保存する必要がある。それ以外のレジスタについては、C言語の処理系によって保存しなければならないかどうかは異なる。MS-Cでは、レジスタ変数として、SIレジスタ、DIレジスタを使用するので、関数内部で使う場合にはこれらを保存しなければならない
- そのほか、各処理系のマニュアルに指示されている約束事を守る。たとえばMS-Cでは、関数名の先頭に「_」を付けなければならない

C言語では、引数をスタックに積んでから関数を呼び出すために、アセンブラルーチン上では、スタックを介してそれらの値を受け取ることができます。その際、通常はBPレジスタをフレームポインタとして使用します。**図7.14**に、一般的な引数を受け取る手順とCのプログラムが関数を呼び出し、BPレジスタをプッシュしたときの、スタックフレームの様子を示しましょう。このときのスタックポインタの値を、BPレジスタに代入してフレームポインタとして使用します。



```
PX      EQU     4
PY      EQU     6
COL     EQU     8

        PUSH    SI
        PUSH    DI
        PUSH    DS
        MOV     AX,[BP].PX
        MOV     BX,[BP].PY
        CALL    COMPADR
        MOV     BX,[BP].COL
        MOV     CX,NPLANE
IF MACHINE EQ PC9801
BITSET: MOV     DS,CS:[SI]
        INC     SI
        INC     SI
ENDIF ; PC9801
IF MACHINE EQ FMR60
        MOV     AX,1B
```

図7.14　アセンブラルーチンがC言語から引数を受け取る手段とスタックの様子

ここで作成するgrplot関数プログラムのように、grplot(x,y,c)のように呼び出す場合には、Xは[BP+4]に、Yは[BP+6]にそれぞれ置かれています。ただし、これはスモールモデル(関数がNEARコールされる)の場合であり、ラージモデル(関数がFARコールされる)の場合には、リターンアドレスが2ワードになるため、引数の位置はスモールモデルより2バイトずつ高位になります。また、Cに戻る際に関数値を返すには、int型などにはAXレジスタを使うのが普通です。AXレジスタに値を入れて戻れば、その値が関数値として使用されます。これらの約束事は、C言語の処理系によって異なる可能性がありますので、詳しくは各処理系のマニュアルを参照してください。

ところで、リスト7.11に示したアセンブリ・ソースファイルは、MS-Cに用意されている、アセンブリ・ソースファイル出力機能を利用して作成したものです。この機能を利用して出力されたファイル(.ASM)は、アセンブル可能な形式のアセンブリ・ソースファイルであり、中身はありませんが、関数の入り口および出口の処理を含んでいますので、ソースファイルにプログラムの中身を加えていけば、入り口や出口の処理を書く手間が省けます。

そのためには、まずC言語で**リスト7.12**のように関数の枠(外側だけで中身なし)だけのソースファイル(ここでは「graphic2.c」)を作ります。

```
/*
 * graphic2.c
 */

#include "graphic.h"

int gropen(void)
{
        return 0;
}

void grclose(void)
{
}

void grplot( unsigned short x,    外側だけで中身のない関数の枠を作る
             unsigned short y,    (この例は「GRAPHIC2.ASM」を作るためのもの)
             unsigned short c )
{
        dummy(); ……………置き換える場所を特定するためのダミーの関数呼び出し
}
```

リスト7.12　関数の枠だけのソースプログラム graphic2.c

このソースファイルを、アセンブリ・ソースファイルを出力させるためのオプションを付けてコンパイルします。**図7.15**にその実行例を示します。

このオプションによりアセンブラのソースファイルを出力することを指定する

```
A>CL -FaGRAPHIC2.ASM -c GRAPHIC2.C↵
Microsoft (R) C Optimizing Compiler Version 5.10
Copyright (c) Microsoft Corp 1984, 1985, 1986, 1987, 1988. All rights reserved.

GRAPHIC2.C

A>
```

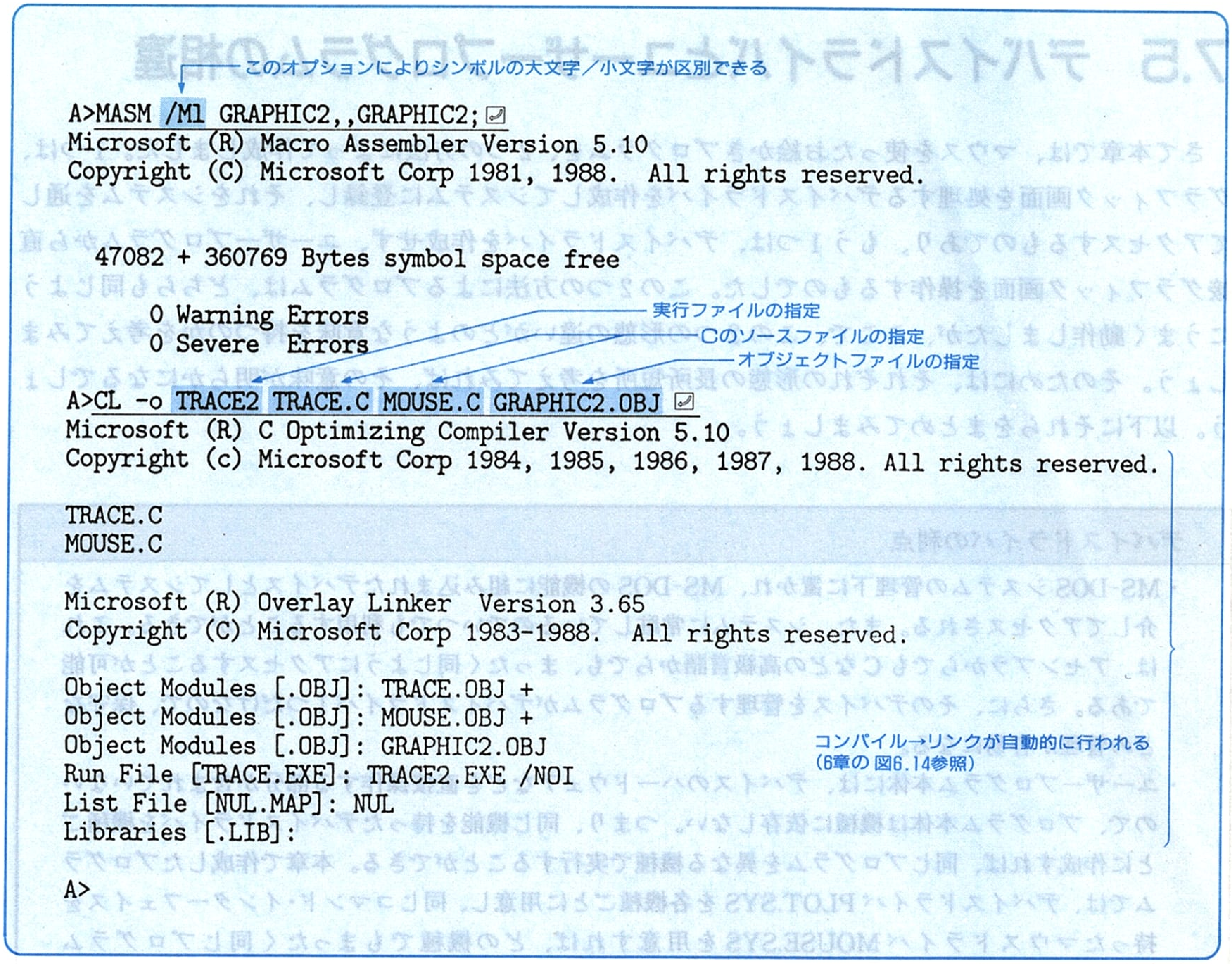
図7.15　Cプログラムからアセンブリ・ソースファイルを出力する実行例

なお、リスト7.11に示したソースファイルGRAPHIC2.ASMには、Cプログラムによって出力された部分とそれに書き加えた部分とを区別して示してありますので参照してください。

ところで、MS-Cの場合、例のCLコマンドを使ってコンパイル&リンクを行うには注意が必要です。CLコマンドはリンカLINKを/NOIオプション付きで呼び出します。これはLINKのバージョン3.0から加わった機能で、シンボルの大文字と小文字を区別することを指定するものです。アセンブラもリンカもデフォルトではそれらを区別せず、すべて大文字に変換して処理してしまうため、アセンブラで書いた関数(小文字の名前)をアセンブルすると、リンクの際に未定義エラーとなってしまうのです。たとえば、ここで作成したGRAPHIC2プログラムの場合、アセンブラ側ではソースは小文字でも「_GRPLOT」と大文字で宣言したことになり、これをC言語側では「_grplot」として呼ぶために、外部参照を解決できなくなってしまいます。

この問題を避けるには、CLコマンドを使わずに、通常のやり方でLINKを実行する方法もあります(/NOIオプションは付けない)。この場合は、アセンブラ側もC言語側も大文字として扱われますので、リンクは矛盾なく行われます。ただし、大文字と小文字で異なる関数(シンボル)を定義したモジュールを含むライブラリがあると誤動作しますので、あまりお勧めできません。

なお、MASMもバージョン3.0からは、シンボルの大文字／小文字を区別することができるようになりましたので、この問題は解決できます。具体的には**図7.16**に示すように、「/ML」オプションを付けてアセンブラを実行します。このオプションを付けた場合は、小文字のシンボルは小文字として、オブジェクトファイルが作られます。



```
A>MASM /Ml GRAPHIC2,,GRAPHIC2;↵
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


  47082 + 360769 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

A>CL -o TRACE2 TRACE.C MOUSE.C GRAPHIC2.OBJ ↵
Microsoft (R) C Optimizing Compiler Version 5.10
Copyright (c) Microsoft Corp 1984, 1985, 1986, 1987, 1988. All rights reserved.

TRACE.C
MOUSE.C

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Object Modules [.OBJ]: TRACE.OBJ +
Object Modules [.OBJ]: MOUSE.OBJ +
Object Modules [.OBJ]: GRAPHIC2.OBJ
Run File [TRACE.EXE]: TRACE2.EXE /NOI
List File [NUL.MAP]: NUL
Libraries [.LIB]:

A>
```

このオプションによりシンボルの大文字／小文字が区別できる

実行ファイルの指定

Cのソースファイルの指定

オブジェクトファイルの指定

コンパイル→リンクが自動的に行われる
（6章の 図6.14参照）

図 7.16　CL コマンドによるコンパイル＆リンクの実行例

本項のプログラム例は、C プログラムからアセンブラで作成した関数を呼び出す手順のよい参考になるでしょう。このように、C プログラムから呼び出すアセンブラのルーチンを書くには、ここで利用したような、アセンブリ・ソースファイルを出力する機能を持つ処理系を使えば、比較的簡単に作成できると思います。

7.5　デバイスドライバとユーザープログラムの相違

さて本章では、マウスを使ったお絵かきプログラムを、2つの方法によって作成しました。1つは、グラフィック画面を処理するデバイスドライバを作成してシステムに登録し、それをシステムを通してアクセスするものであり、もう1つは、デバイスドライバを作成せず、ユーザープログラムから直接グラフィック画面を操作するものでした。この2つの方法によるプログラムは、どちらも同じようにうまく動作しましたが、ここで、この2つの形態の違いがどのような意味を持つのかを考えてみましょう。そのためには、それぞれの形態の長所短所を考えてみれば、その意味が明らかになるでしょう。以下にそれらをまとめてみましょう。

デバイスドライバの利点

・MS-DOSシステムの管理下に置かれ、MS-DOSの機能に組み込まれたデバイスとしてシステムを介してアクセスされる。また、システムに常駐しているのでいつでも利用することができる。これは、アセンブラからでもCなどの高級言語からでも、まったく同じようにアクセスすることが可能である。さらに、そのデバイスを管理するプログラムがデバイスドライバ1つだけなので、保守などの管理が容易になる。

・ユーザープログラム本体には、デバイスのハードウェアなどを直接操作する部分が含まれていないので、プログラム本体は機種に依存しない。つまり、同じ機能を持ったデバイスドライバを機種ごとに作成すれば、同じプログラムを異なる機種で実行することができる。本章で作成したプログラムでは、デバイスドライバPLOT.SYSを各機種ごとに用意し、同じコマンド･インターフェイスを持ったマウスドライバMOUSE.SYSを用意すれば、どの機種でもまったく同じプログラムTRACE.EXEが動作する。つまり、ソースファイルの書き換えや、アセンブルやコンパイルをやり直すことなく、同じ実行可能プログラムをそのまま動作させることが可能なのである。

デバイスドライバの欠点

・システムを介してデバイスの操作を行うため、若干のオーバーヘッドがあり、その分の処理速度が低下する。とくに本章のお絵かきプログラムの場合は、システムを介するために、座標値をわざわざ文字列に変換しなければならないので、かなりのロスになる。

・デバイスドライバを書くのは、デバイスを直接アクセスするために必要な部分以外に、OSとのインターフェイスの部分を書かなければならないため、かなりやっかいな仕事になる。本章で作成したデバイスドライバを見ても、それがかなりの部分を占めている。

デバイスドライバとせず、ユーザープログラムで処理する利点

- ・ユーザープログラムから直接デバイスを操作すれば、オーバーヘッドが少ないために、アクセスが高速になる(間にシステムを介さない分、アクセスが速いのは当然)。
- ・デバイスを直接操作するユーザープログラムを書くのは、OSとのインターフェイス部分を書く必要がないため、デバイスドライバを書くより簡単である。

デバイスドライバとせず、ユーザープログラムで処理する欠点

- ・異なる機種上では、ほとんどの場合、そのままのプログラムでは動作しない(MS-DOSの管轄外のデバイスをアクセスしているのだから当然)。また、コンソールなどのMS-DOSが認識しているデバイスでも、それを直接アクセスする場合は、機種によって方法が異なるために動作しない。異なる機種で同じプログラムを動作させるには、デバイスを操作する部分を書き直し、アセンブル、コンパイル、さらにリンクをやり直さなければならない。例題のプログラムではGRAPHIC2.ASMを書き直し、アセンブル、リンクをやり直す必要がある。
- ・異なるプログラミング言語からそのデバイスをアクセスするには、そのプログラミング言語で、あるいはそのプログラミング言語とインターフェイスできるほかのプログラミング言語で、そのアクセスプログラムを書き直さなければならない。つまり、言語ごとにデバイス操作のプログラムを書かなければならない。

さて、両者にはこのような違いがありますが、一般に、高速性を要するもの、たとえばグラフィックスや高速通信などで、特定の機種でのみ動作すればよく、ほかの機種で動作する必要のないものなどは、ユーザープログラムから直接デバイスを操作することになるでしょう。しかし、これはOSの約束に従っていないものであり、あまり推奨されたやり方ではありません。デバイスの操作は、なるべくデバイスドライバの形態で行うべきです。その方が仕様を変更したり、拡張したり、移植したりする場合に便利です。

3章で述べたように、OSには異なる機種間で同じプログラムを動作させるという役割があります。ところが市販のソフトウェアの中には、OSはMS-DOSでも、特定機種専用などというものも少なくありません。これらのソフトは、MS-DOSのファイルシステムなどの機能を利用しているだけで、コンソールやグラフィックスなどは、その機種のハードウェアを直接操作しています。このようなソフトでも、MS-DOSのファイルシステムを利用することにより、従来のOSやBASICのファイルシステムでは実現できなかった、階層ディレクトリの恩恵にあずかったり、入出力ファイルを、ほかのMS-DOSのプログラムで処理することなど、格段に柔軟な取り扱いが可能になります。

とはいっても、同じプログラムを異なる機種間で動作させるというOSの役割を考えると、やはり理想的にはグラフィックスなども、OSのシステムコール、あるいはデバイスドライバによってMS-DOSがサポートすることが望まれます。Microsoft-Windowsなどはこのような考え方のもとに開発され、次世代の強力マシンと組み合わせての一層の普及が期待されています。

APPENDIX

ここでは、2章のクラスタ追跡の実験に使った数字の羅列ファイル「TEST」を作成するためのプログラム「mk012.c」、および、3章の「フィルタ」の解説のところで紹介したテキストファイル中の英大文字↔英小文字の変換プログラムのソースプログラム「ulconv.c」を示します。なお、コンパイルやリンクの方法などについては、6章を参照してください。

```
/*
 * mk012.c
 */

#include <stdio.h>
#define MAXDATA (2500/4)

void    main(int argc, char *argv[])
{
        int     i;
        FILE    *fp;

        if (argc != 2)
                exit(1);
        fp = fopen(argv[1], "w");
        if (fp == NULL) {
                fprintf(stderr, "Can't create %s¥n", argv[1]);
                exit(1);
        }
        for (i = 0; i < MAXDATA; i++)
                fprintf(fp, "%03d ", i);
        fclose(fp);
}

/* end of mk012.c */
```

Microsoft C、Turbo C、Quick C でコンパイル可能。プログラム名を「MK012」とすると、実行可能プログラム「MK012.EXE」を作成する。完成したら、

A>MK012 ファイル名↵

を実行することによって、テスト用ファイルが作成される。

ここでは2500バイトにしている

#define MAXDATA (2500/4)……ファイルのサイズを決める定数

main(int argc, char *argv[])……コマンドラインのパラメータの数とその文字列

if (argc != 2)……コマンドラインは「A>MK012 ファイル名↵」なので、パラメータは2つあるはず

fp = fopen(argv[1], "w");……「ファイル名」のファイルをオープンする

exit(1);……終了コード「1」を返してこのプログラムを終了する

fprintf(stderr, … exit(1); } ……オープンできない場合は、エラーメッセージを表示する

for … fprintf(fp, "%03d ", i); ……「000␣001␣002␣……」という文字列を書き込んでいく

fclose(fp);……ファイルをクローズする

<数字羅列ファイル作成プログラム mk012.c>

```
/*
 * ulconv.c
 */
#include <stdio.h>
#include <ctype.h>
#include <jctype.h>

typedef enum {LOWER, UPPER} flag;

void ulconv(flag const f, FILE * const fp);
```

typedef enum {LOWER, UPPER} flag;……変換の種類を表すための列挙型定義

(次ページに続く)

```
void main(int argc, char const *argv[]) {
        FILE *fp;
        flag f = UPPER; ・・・・・・・・・・・・スイッチの指定がなければ小文字→大文字の変換処理へ

        while (--argc > 0) { ・・・・・・・・・・・・コマンドラインのパラメータをチェックしていく
                if ((*++argv)[0] != '-') ・・・・・・・・・・・・1文字目が「ー」でないならばファイル名である
                        break;
                switch (toupper((*argv)[1])) {
                case 'L':
                        f = LOWER; break; ・・・・・・・・・・・・大文字→小文字変換
                case 'U':
                        f = UPPER; break; ・・・・・・・・・・・・小文字→大文字変換
                default:                                                        存在しないスイ
                        fprintf( stderr, "ulconv:illegal switch %c¥n",          ッチを指定した
                                          (*argv)[1] );                          場合標準エラー
                                                                                 出力へエラーメ
                                                                                 ッセージを出力
                                                                                 する
                        exit(1); ・・・・・・・・・・・・リターンコード1(異常終了)で終了する
                }
        }
        if (argc == 0) { ・・・・・・・・・・・・ファイル名の指定がなければ標準入力
                ulconv(f, stdin);
        } else if ((fp = fopen(*argv, "r")) == NULL) { ・・・・・・・・・・・・ファイルをオープンする
                perror(*argv);  } オープンできなかった場合、エラーメッセージを出力して
                exit(1);        } リターンコード1で終了する
        } else {
                ulconv(f, fp); ・・・・・・・・・・・・ファイルを変換する
                fclose(fp); ・・・・・・・・・・・・ファイルをクローズする
        }
        exit(0); ・・・・・・・・・・・・正常終了で終了する
}

void ulconv(flag const f, FILE * const fp) {
        register int ch;

        while ((ch = getc(fp)) != EOF) { ・・・・・・・・・・・・EOFを読むまで繰り返す
                if (iskanji(ch)) { ・・・・・・・・・・・・漢字の場合の処理
                        putchar(ch); ・・・・・・・・・・・・そのまま出力
                        if ((ch = getc(fp)) == EOF) {
                                fputs("Incomplete KANJI character¥n", stderr);
                                break;
                        }                                                漢字の2バイト目を読み
                                                                         込む。EOFならばエラ
                                                                         ーメッセージを標準エラ
                                                                         ー出力する
                } else if (isalpha(ch)) { ・・・・・・・・・・・・英字の場合
                        switch (f) {
                        case LOWER:
                                if (isupper(ch)) ch = tolower(ch);
                                break;                                   指定に従って
                        case UPPER:                                      変換する
                                if (islower(ch)) ch = toupper(ch);
                                break;
                        }
                }
                putchar(ch);
        }
}
/* end of ulconv.c */
```

＜大文字↔小文字変換プログラム ulconv.c＞

おわりに

今回の改訂では、完全とはいかないまでも、内容が格段に進歩した、今まで以上の解説書に仕上ったのではないかと自負しています。

初版本にも書いたのですが、どのページにも省けない重要な情報がぎっしり書かれている中で、7章の図7.12の〝作品〟「虫取り少年とひまわり」の素朴なタッチは、『あの頃の夏の日のぼく』にタイムスリップさせてくれる、本書の中で唯一気が抜ける部分です。「もっとカッコいい絵にしましょうか」という話も出たのですが、そんなことから、そのままにしておいてもらったいきさつがありました。

さて、現在はまさにMS-DOS全盛の時代ですが、MS-DOSが登場してから、MS-DOSの機能を十分に活用し、使いこなしたアプリケーションプログラムが広く世の中に出てくるまでには、少なくても3～4年以上もの熟成期間を必要としました。MS-DOSより数段高度なOS/2では、それ以上の期間が必要かもしれません。「はじめに」でも述べたように、MS-DOSは今後ともかなり長期間にわたって使われ続ける見とおしであり、MS-DOSのプログラミングの知識は、その必要性がますます高まる一方です。そして、MS-DOSシステムの本格的な理解と応用こそ、WINDOWSや次世代のOSであるOS/2を発展させ、ひいてはパーソナルコンピュータのソフトウェアシーンを発展させる言動力になると信じています。本書がその重要な役目の一端を担い、本書に接した人たちによって、さらに進化したソフトウェアの世界が展開されることを願っています。

なお、本書を執筆するにあたり、以下の文献を参考にしました。

- 『標準MS-DOSハンドブック』　アスキー出版局編著　アスキー出版局発行　1984
- 『MS-DOSプログラマーズハンドブック』『MS-DOSプログラミングテクニック』
  『MS-DOS 3.1ハンドブック』　アスキー書籍編集部編著　アスキー出版局発行　1985
- 『MS-DOSエンサイクロペディア Volume 1』『同 Volume 2』
  マイクロソフトプレス編　アスキー出版局発行　1989
- 『応用C言語』　三田典玄著　アスキー出版局　1988
- 『はじめて読む8086』『はじめて読むMASM』　蒲地輝尚著　アスキー出版局発行　1987／1988
- 『PC-9801シリーズ テクニカルデータブック』『同 増補改訂版』
  テクライト編　アスキー出版局発行　1986／1988
- 『富士通FMRシリーズ 徹底解析マニュアル 増補改訂版』　インタープログ編　BNN発行　1987

および、以下のマニュアル

- MS-DOS バージョン2.0／3.1／3.3
- マイクロソフト マクロアセンブラ バージョン5.1
- マイクロソフト C オプティマイジング コンパイラ バージョン5.1
- Turbo C バージョン2.0

索　引

A
ADDDRV …… 123, 170, 171
ANSI-C …… 295
argc …… 290
argv …… 290
ASCII コード …… 153
ASCII ファイル …… 80
ASCIZ 文字列 …… 153, 154, 299
ASSUME …… 252, 253
ATTRIB …… 11
AUTOEXEC.BAT …… 125, 145, 149, 311
AUX …… 144, 163

B
BIOS …… 118
BPB …… 166, 167, 168
BREAK …… 123
BUFFERS …… 123

C
CODEVIEW …… 272, 283, 313, 314
COMMAND.COM　117, 124, 126, 128, 130, 134, 144, 148
COM 形式 …… 154, 248, 252, 256, 260, 333
COM ファイル …… 13, 23, 24, 25
CON …… 138, 144, 163, 164
CONFIG.SYS …… 44, 66, 90, 121, 122, 149, 164, 170, 331
COOKED モード …… 219
COPY …… 322
CP／M …… 116
CR …… 197, 298
CREF …… 245, 268
Ctrl-C …… 79, 94, 123, 197, 198

D
DEBUG …… 70, 272, 274, 282
DELDRV …… 170
DEVICE …… 123, 168
DOS …… 116

E
EMS …… 171
END …… 253, 271, 333
ENDS …… 252
EOF …… 208
EXE2BIN …… 13, 24, 246, 270, 330
EXE 形式 …… 248, 252, 256, 333
EXE ファイル …… 13, 25, 270
EXIT …… 146, 149
exit …… 291
EXTRN 宣言 …… 254

F
FAT …… 40, 42, 53, 56, 58, 79, 84, 95
FAT　ID …… 70, 73, 85, 167
FAT エントリ …… 55, 84, 86, 88, 96, 98
FAT 数 …… 70
FAT 領域 …… 55, 68, 69, 72
FCB …… 38, 42, 44, 48, 55, 56, 137
FILES …… 122
FIND …… 140
fprintf …… 297

H
hidden ファイル …… 11

I
INT　20H …… 189
INT　21H …… 189, 192, 293
INT　22H …… 191
INT　23H …… 191
INT　24H …… 191
INT　25H …… 189, 190
INT　26H …… 190
INT　27H …… 190
IOCTL …… 219
IO.SYS …… 114, 117, 120, 126, 132, 158
IPL …… 118, 120
I／O コントロール …… 219
I／O リクエストコマンド …… 336, 337, 338

L
LF …… 197, 298
LIB …… 243
LINK …… 13, 22, 28, 234, 246, 269
LINT …… 296

M

main …… 290, 291
MAKE …… 245, 301
MAPSYM …… 280
MASM …… 13, 28, 229
MORE …… 140
MS-DOS …… 107
MSDOS.SYS …… 117, 126, 128, 131, 132, 147, 150, 156
MS-DOS システム …… 114, 126

O

ORG …… 253
OS …… 112, 116, 129

P

PATH …… 135, 145
printf …… 292, 293
PRN …… 144, 163, 164
PROC …… 254, 300, 337
PROMPT …… 145
PSP …… 151, 152, 154, 155, 290
PUBLIC 宣言 …… 254, 280

R

RAM ディスクドライバ …… 171
RAM 領域 …… 120
RAW モード …… 219
RS-232C …… 162, 163

S

SEGMENT …… 251, 252
SET …… 145, 151
SHELL …… 124, 149
SORT …… 140
STACK …… 252
STACK 属性 …… 252
START …… 253
SYMDEB …… 70, 272, 274, 280, 282
SYS …… 119

T

TYPE …… 31

U

UNIX …… 116

12 ビット FAT …… 90
16 ビット FAT …… 89, 90
#define …… 293
#endif …… 293
#ifdef …… 293
.. …… 63
. …… 63
.BAT …… 135
.COM …… 135, 248
.EXE …… 135, 248
/P オプション(COMMAND.COM の) …… 124, 149
< …… 136, 140
> …… 136, 140
>> …… 136, 140
¥n …… 298
| …… 140

ア

アーカイブファイル …… 58
空きマーク …… 49, 93
空き領域 …… 49, 53, 54
アセンブラ …… 28, 289
アセンブリ言語 …… 229
アセンブル …… 15, 18, 22, 245, 267
アセンブルエラー …… 19, 21
アトリビュート …… 11, 51
アドレス …… 232, 234
アルゴリズム …… 272
インタラプト …… 156
ウォッチ機能 …… 315
ウォッチポイント機能 …… 315
エスケープシーケンス …… 162
エラー …… 185, 295
エラーチェック …… 185
エラーハンドル …… 144
エラーメッセージ …… 20, 21, 123, 144, 297
オブジェクト形式 …… 154
オブジェクトファイル …… 20, 28, 248, 267, 270
オプションコマンド …… 124
オープン …… 38, 44, 48, 129, 140
オペレーティングシステム …… 107
親プロセス …… 148, 150, 223

カ

階層ディレクトリ …… 58
外部プログラム …… 127, 133, 134
外部割り込み …… 157
改ページコード …… 197
書き込み …… 44, 48, 51, 54

隠しファイル …… 11, 25, 75
環境 …… 145, 148
環境セグメントアドレス …… 151, 154
環境変数 …… 135, 145, 146, 147
環境変数名 …… 145
環境文字列 …… 145, 146, 152, 154
擬似命令 …… 232, 251
起動 …… 118, 125, 149
機能の呼び出し …… 37
キーボード …… 162
基本ソフトウェア …… 110
逆アセンブル …… 277
キャラクタ型デバイス …… 160, 162, 170, 322
キャリッジリターン …… 197
共有ライブラリ …… 115
空白の領域 …… 89
クラスタ …… 40, 54, 55, 56, 69, 80, 88, 96, 98
クラスタサイズ …… 89
クラスタ番号 …… 35, 37, 40, 82, 89
クラスタ領域 …… 68
グラフィック画面 …… 323
クリエイト …… 48
クローズ …… 44, 51, 55, 129, 136, 140
クロスリファレンスファイル …… 245, 267, 284
クロスリファレンス・リスティングファイル …… 245
高級言語 …… 229
構造化機能 …… 251
互換性 …… 69, 90, 177, 295
子プロセス …… 136, 148, 150, 223
コマンド …… 118, 139
コマンドパケット …… 336, 337
コマンド・プロセッサ …… 124, 126
コマンド名 …… 147, 152
コマンド文字列 …… 136
コマンドライン文字列 …… 290
コンソール …… 31, 138, 162
コンソールドライバ …… 219
コンパイル …… 311

サ

サイド …… 67
作成 …… 49
サーチパス …… 146
サブディレクトリ …… 58, 59, 69, 99
サブディレクトリ名 …… 59
システムコール …… 17, 28, 114, 128, 158, 177, 189, 293, 296
システム属性 …… 25, 305
システムファイル …… 11, 25, 58, 75
システムマクロ …… 238
システム予約領域 …… 68, 69, 118
実行開始アドレス …… 155
周辺装置の管理 …… 131
常駐 …… 149, 171, 173
書式変換 …… 292
シリンダ …… 67
シングルタスク …… 139, 337
シンボリックデバッガ …… 280, 282
シンボル …… 232, 254, 356
シンボルマップファイル …… 280, 282
スイッチ …… 269
スキップセクタ …… 54, 55
スタック …… 23, 25
スタックセグメント …… 271
ストラテジ・エントリアドレス …… 335
ストラテジルーチン …… 337
セクタ …… 40, 42, 67, 69, 80
セクタサイズ …… 42, 84
セクタ数 …… 40
セクタ長 …… 40
セグメント …… 154, 155, 251
セグメントオーバーライド・プリフィックス …… 333
セグメントベース …… 154
セグメント方式 …… 154
セグメントレジスタ …… 154, 155
属性 …… 11, 51, 75, 76, 77, 252
ソースコードデバッガ …… 283
ソースファイル …… 267
ソースリスティングファイル …… 267
ソフトウェア割り込み …… 156, 158

タ

ターゲットプログラム …… 152, 274, 282
ターミナル装置 …… 159
通常のファイル …… 11, 25, 76
ツール …… 13
ディスク …… 67
ディスク管理 …… 107
ディスクキャッシュ …… 171
ディスクドライブ …… 162
ディスク・バッファリング …… 42, 51, 55
ディスクフォーマット …… 67, 70
ディスクフル …… 54
ディスクリセット …… 79
ディスプレイ …… 31, 162
ディレクトリ …… 35, 55, 79, 99
ディレクトリエントリ …… 35, 36, 59, 63, 75
ディレクトリ領域 …… 55, 68, 73
テキスト形式 …… 20, 298, 322

テキストファイル……31, 80, 220
データ領域……58, 68, 85, 98, 119
手続き……254
デバイス……136
デバイス属性……335
デバイスドライバ……123, 158, 163, 170, 320, 322, 358
デバイスファイル……136, 163
デバイスファイル名……168, 324
デバイスヘッダ……333
デバイス名……164, 324
デバッガ……70, 246, 272, 274, 314
デバッグ作業……13, 25, 246, 271, 278
テンポラリファイル……139, 140
統合プログラミング環境……313
トップダウン法……250
トラック……40, 67
トレース……277

ナ

内部割り込み……157
入出力……140

ハ

バイト数……40, 42
バイナリ形式……322
パイプ……136, 139, 140, 144, 152
バッチコマンド……146
バッチファイル……146
バッファ……42, 51, 53, 63, 66, 197
バッファ数……123
バッファ領域……66
バッファリング……42, 51, 63, 66, 79, 297
ハードウェア割り込み……157
ハードディスク……25, 89
パブリックシンボルマップファイル……280
パラメータ……147, 290
引数……290
標準エラー出力……144, 211, 297
標準出力……137, 138, 144, 211
標準ディスクフォーマット……67
標準デバイス……164
標準デバイスドライバ……168
標準入出力……136, 137, 138, 139, 140, 142, 148
標準入力……137, 138
標準ファイルハンドル……144
標準フォーマット……69, 70
ファイル……31, 58, 69, 91, 99, 129
ファイルアクセス・コントロール……204, 205
ファイルアロケーション・テーブル……40
ファイルオープン……35
ファイル管理……82, 131
ファイル記述子……38
ファイルコントロール・ブロック……38
ファイルサイズ……35, 37
ファイル作成日時……35
ファイルシステム……31, 35, 38, 79, 108, 118, 131, 165
ファイル数……122
ファイル属性……11, 17, 28, 58, 99
ファイルタイプ……136, 248, 267
ファイルの末尾……208
ファイルの読び出し……39
ファイルハンドル……38, 44, 48, 56, 122, 136, 138, 144, 296
ファイル名……35, 93, 95
ファンクション……177
ファンクション 0AH……197, 198
ファンクション 00H……189
ファンクション 02H……203, 211
ファンクション 05H……197, 198, 216
ファンクション 09H……211
ファンクション 3CH……205
ファンクション 3DH……37, 203, 215
ファンクション 3EH……203, 205, 215, 216
ファンクション 3FH……203, 215
ファンクション 31H……190
ファンクション 4AH……222
ファンクション 4BH……222
ファンクション 4CH……191, 197
ファンクション 4DH……198
ファンクション 40H……205, 211, 215
ファンクション 42H……205
ファンクション 43H……220
ファンクション 44H……219
ファンクションリクエスト……17, 37, 42, 44, 48, 158, 177, 188, 196
フィルタ……140, 141
フォーマット……70
復帰改行コード……298
フラッシュ……56
不良マーク……55
プリンタ……160, 162, 163
ブレークポイント……273, 275, 280
プログラミング言語……229
プログラムセグメント・プレフィックス……151
プロシージャ……254
プロセス……148, 150
プロセス管理……28, 223
ブロック型デバイス……160, 162, 165
プロトタイプ宣言……296, 301

プロンプト …… 125
分割コンパイル …… 300
分岐命令 …… 231
補助装置 …… 162
補助入出力装置 …… 163
ボトムアップ法 …… 250
ボリュームラベル …… 58

マ

マウスドライバ …… 321
マクロ …… 237
マクロアセンブラ …… 13, 234, 251
マクロ機能 …… 237, 240
マクロ定義 …… 234
マクロパラメータ …… 237, 238
マクロ名 …… 234
マクロ呼び出し …… 234
マルチタスク …… 150, 337
未バックアップファイル …… 58
メイクファイル …… 301
メディア …… 40
メディアチェック …… 166, 168
メモリ管理 …… 28, 131, 150, 223
文字ファイル …… 80
モジュール …… 232, 240, 245, 254
モジュール別プログラミング …… 242, 256, 260, 300

ヤ

ユーザープログラム …… 127, 131
読み出し …… 48
読み出し/書き込み …… 48
予約セクタ …… 72, 73
予約領域 …… 72

ラ

ライブラリ …… 112, 243, 283, 284, 289, 295
ライブラリアン …… 243
ライブラリ関数 …… 289, 294, 296, 299
ライブラリファイル …… 284
ライブラリマネージャ …… 243
ラインフィード …… 197
ラベル …… 230, 231, 232, 234, 246, 254
リアルタイムクロック …… 162
リスティング …… 284
リスティングファイル …… 20, 245
リダイレクト …… 38, 136, 144, 214, 297
リードオンリーファイル …… 11, 25, 75
リピート擬似命令 …… 238
リブートパス …… 149
リロケーション情報 …… 270
リロケータブル …… 154, 155
リロケータブル・アセンブラ …… 232, 251
リロケータブル・オブジェクト …… 232, 234, 269
リロケータブル・オブジェクトファイル …… 245, 300
リロケータブル・マクロアセンブラ …… 229
リロケート …… 270, 271
リロケート機能 …… 232
リロケート操作 …… 155
リンカ …… 22, 28, 234, 246, 269
リンク …… 232, 234, 269, 284, 311
ルートディレクトリ …… 58, 69
論理セクタ番号 …… 70, 72, 82, 87
論理セグメント …… 155

ワ

割り込み …… 156
割り込みエントリアドレス …… 335
割り込みテーブル …… 156
割り込みベクタテーブル …… 125

MS-DOS、Microsoft C Compiler、Microsoft Quick C Compiler は、米国 Microsoft 社の商標です。
IBM、IBM PC、PC-DOS は、International Buisnedd Machines 社の登録商標です。
CP／MCP／M-86 は、米国 Digital Reserch 社の登録商標です。
UNIX オペレーティングシステムは、AT&T のベル研究所が開発し、AT&T がライセンスしています。
Turbo C は、米国 Borland International 社の登録商標です。

●執筆協力
蒲地輝尚

応用 MS-DOS　改訂新版

アスキー・ラーニングシステム③応用コース

1986年7月31日　初版発行
1989年12月21日　第2版第1刷発行
1990年6月11日　第2版第4刷発行
定価2,300円(本体2,233円)

著　者　村瀬康治
発行者　塚本慶一郎
発行所　株式会社アスキー
〒107-24　東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
TEL (03)486-7111（大代表）
情報 TEL (03)498-0299（ダイヤルイン）
出版営業部 TEL (03)486-1977（ダイヤルイン）

本書は著作権法上の保護を受けています．本書の一部あるいは全部について（ソフトウェア及びプログラムを含む），株式会社アスキーから文書による許諾を得ずに，いかなる方法においても無断で複写，複製することは禁じられています．

制　作　株式会社 GARO
印　刷　株式会社 加藤文明社印刷所

編　集　　　小栗葉子

ISBN 4-7561-0018-X　　Printed in Japan

● 11136／11403／11491

ALS
ASCII Learning System

定価2,300円（本体2,233円）

ISBN4-7561-0018-X C3055 P2300E